

AFRL-IF-RS-TR-2005-319
Final Technical Report
September 2005



PROFILER-2000: ATTACKING THE INSIDER THREAT

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J765

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-319 has been reviewed and is approved for publication

APPROVED: /s/

DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: /s/

JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE SEPTEMBER 2005	3. REPORT TYPE AND DATES COVERED Final Apr 00 – Jun 04	
4. TITLE AND SUBTITLE PROFILER-2000: ATTACKING THE INSIDER THREAT			5. FUNDING NUMBERS C - F30602-00-2-0528 PE - 62301E PR - IAST TA - 00 WU - 06	
6. AUTHOR(S) R. A. Maxion, K. M. C. Tan, S. S. Killourhy and T. N. Townsend				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University 5000 Forbes Avenue Pittsburgh Pennsylvania 15213-3890			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFED 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2005-319	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Deborah A. Cerino/IFT/(315) 330-1445/ Deborah.Cerino@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) The Profiler project was concerned with fundamental theoretical and measurement issues in the field of anomaly/intrusion detection, particularly as directed at the problem of insiders. Major issues addressed were: scientifically sound foundations for divers anomaly detectors; tools and methods for assessment of detector effectiveness; and controlled benchmark data sets for testing. Major accomplishments of the project were: determining how the interaction between the architectural aspects of a detection algorithm, such as detection mechanism and coverage, can result in unanticipated vulnerabilities that allow an adversary to undermine the detector; production of calibrated test data sets; and rigorous assessment and error analysis of an anomaly detector in an insider-threat environment.				
14. SUBJECT TERMS Anomaly Detection, Insider Threat, Profiling, Benchmarks			15. NUMBER OF PAGES 151	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

Overview.....	1
List of Deliverables.....	3
Task 1 Descriptive characteristics of data.....	5
Task 2 Impact of data characteristics on detection performance.....	6
Task 3 Architectural aspects and impacts of anomaly-detection algorithms.....	7
3.1 Introduction.....	7
3.2 Approaches to undermining anomaly detectors.....	9
3.3 Detection coverage of an anomaly detector.....	10
3.4 Deploying Exploits and Sensors.....	13
3.5 Where and how an attack manifests in the data.....	13
3.6 Manipulating the manifestation; modifying exploits.....	17
3.7 Evaluating the effectiveness of exploit modifications.....	21
3.8 Discussion	22
3.9 Related work	24
3.10 Conclusion.....	24
Task 4 Interaction effects: data characteristics vs. detector architecture	26
4.1 Introduction.....	26
4.2 Hypothesis and approach	27
4.3 Examining the mathematical model of naive Bayes	28
4.4 Empirical study using synthetic data	33
4.5 Mitigating the effect of NBSCs	37
4.6 Conclusion	39
Task 5 Review of facts acquired and lessons learned in developing a science base for anomaly detection.....	40
Task 6 Program code and documentation for the CSD anomaly detector	41
6.1 Introduction.....	41
6.2 Description.....	41
6.3 Usage.....	43
Task 7 Test data suites used in ascertaining the efficacy of the CSD algorithm	47
7.1 Introduction.....	47
7.2 Description.....	47
Task 8 Strengths and weaknesses of the CSD detection algorithm	50
8.1 Introduction.....	50
8.2 Description.....	50
8.3 Evaluation Methodology	52
8.4 Results.....	63
8.5 Discussion.....	64
8.6 Conclusion -a summary of the lessons learned	71
Task 9 Tool for elucidating the sequential dependencies in categorical audit-type data ..	73
9.1 Introduction.....	73
9.2 Description.....	73
9.3 Usage.....	75
Task 10 Tool for creating benchmark datasets that reflect intrinsic environmental structure	76

10.1 Introduction.....	76
10.2 Description.....	76
10.3 Usage.....	77
Task 11 Tool for generating benchmark datasets containing specified regularity	80
11.1. Introduction.....	80
11.2 Description.....	81
11.3 Running Syngen	84
Task 12 Benchmark datasets from enterprise environments.....	86
Task 13 Series of benchmark datasets containing well-specified ground-truth	87
13.1 Introduction.....	87
13.2 Description.....	87
Task 14 Benchmark dataset specifically tailored to address the insider-threat and profiling problems	91
14.1 The dataset.....	91
Task 15 Data-gathering and generating methodologies	96
Task 16 Guidance in the use of benchmark datasets for anomaly detection	97
Task 17 Assessment: Assessing a detector in the insider-threat environment.....	98
17.1 Introduction.....	98
17.2 Problem and approach.....	100
17.3 Related Work	100
17.4 The Naive Bayes Algorithm.....	101
17.5 Data.....	104
17.6 Improvements on previous results.....	104
17.7 Insight into masquerader success	115
17.8 Discussion.....	125
17.9 Conclusion	127
17.10 Appendix: Details of the two-class training experiments	128
Bibliography.....	139

List of Figures

Figure 3.1: The detector coverage (detection map) for stide; A comparison of the size of the detector window (rows) with the ability to detect different sizes of minimal foreign sequence (columns). A star indicates detection.....	12
Figure 3.2: The manifestation of each version of the tracerouteexploit plotted on the detector coverage map for stide, assuming that stide has been configured with a detector window size of 6. Each version of the exploit can be detected by fewer and fewer configurations of stide until the last is invisible.....	22
Figure 4.1: Graph of the relationship between the number of NBSCs and the anomaly score in the mathematical model of the naive Bayes algorithm.....	32
Figure 4.2: Graph of the effect the number of NBSCs has on the percentage of synthetic datasets in which the masquerader block was correctly detected.....	35
Figure 4.3: Graph of the effect of NBSCs on the anomaly score of a masquerade block when there is one nonself user, the commands in the masquerade block never appear in the self training data and appear with a relative frequency of 1.0% in the nonself training data.....	36
Figure 4.4: Graph of the effect of NBSCs on the anomaly score of all 121 masquerade blocks for which the number of nonself users was 10. The commands used in the masquerade blocks appear with in the self training data with frequencies ranging from 0.0% to 1.0% and appear in the nonself training data with the same range of frequencies.....	37
Figure 4.5: Graph of the ROC curves for both naive Bayes (green) and the new masquerade detector fortified with a NBSC detector (black).....	38
Figure 8.1: The desired effect of a controlled injection introducing the foreign-sequence anomaly with foreign internal sequences and rare boundary sequences background data consisting of common sequences only.....	59
Figure 8.3: The incident span. A detector's response to the elements within the incident span are included in the calculation of hit and miss rates.....	63
Figure 8.4: The performance space for the CSD. The detector is completely blind to rare-sequence anomalies	65
Figure 8.5: Similarity calculation between two size-9 sequences, where the foreign element lies at the end of the sequence. The oval shape marks the position of the foreign element in the foreign sequence. The step curve represents the weight contributed by each match. Dotted lines indicate the weight that would have been contributed if an exact match occurred in that position.....	68
Figure 8.6: Similarity calculation between two size-9 sequences, where the foreign element lies in the middle of the sequence. The oval shape marks the position of the foreign element in the foreign sequence. The step curve represents the weight contributed by each match. Dotted lines indicate the weight that would be been contributed if an exact match occurred in that position.....	68
Figure 8.7: Describing a "most similar" foreign sequence.....	70
Figure 17.1: Receiver operating characteristic (ROC) curves for Naive Bayes classifier, with updating, on sequences of length 10 and 100. Superimposed are best results achieved by the various methods described in the text. Solid curve shows sequence length 100; dotted curve shows sequence length 10.....	108

Figure 17.2: Concept drift illustrated by User 10: scattergram representation of commands. Vertical dashed line indicates division between training data (left of line) and testing data. Note regions of test data bearing no resemblance to any portion of training data.....	114
Figure 17.3: User 38: scattergram representation of training data.....	119
Figure 17.4: User 5: scattergram representation of training data.....	120
Figure 17.5: Correlation between use of shared tuples (common to 10 or more users) and masquerader success. To avoid clutter, only selected points are labeled with a user number.....	122

List of Tables

Table 3.1: Elements of methodology for undermining.....	8
Table 3.2: The system calls that implement each of three methods that attempt to achieve the same goal of securing an interactive root account accessible to the attacker.....	18
Table 8.1: The training-data generator. The data generator is a transition matrix where the first column contains the current events and the first row contains the next event. Hence, if the current event is 3 then the probability that the next event is 4 is 0.967200.....	55
Table 8.2: The anomaly detection capability of the CSD detector with respect to the detection of foreign-symbol, foreign-sequence and rare-sequence anomalies.....	63
Table 17.1: Summary results of Naive Bayes detector on sequence lengths 10 and 100.....	106
Table 17.2: Summary results of various Schonlau et al. detectors on sequence length 100.....	107
Table 17.3: Ranking of classification methods, using SEA data configuration and Cost = Misses + 6 * (False Alarms) as a ranking function (based on Schonlau et al. assessment of Uniqueness as their best method -see text for details).....	109
Table 17.4: Summary results of one-class training input; Naive Bayes detector on SEA and 1v49 data configurations.....	116
Table 17.5: Confusion matrix (excerpt): simple Naive Bayes classifier without updating, 1v49 configuration. “Self” is listed down the left column; “non-self” is listed across the top. Numbers in the matrix indicate misses – the number of times the detector erroneously accepted a masquerader (non-self) as self, e.g., User 5 successfully masqueraded 46 times as User 9.....	118
Table 17.6: Ranked percentage of training data covered by tuples common to at least 10 users.....	121
Table 17.7: Summary results of de-tupled data experiments.....	124
Table 17.8: Two-class training input; Naive Bayes algorithm without updating; SEA data configuration; sequence size 100; threshold 1.263.....	129
Table 17.9: Two-class training input; Naive Bayes algorithm with updating; SEA data configuration; sequence size 100; threshold 1.227.....	130
Table 17.10: Two-class training input; Naive Bayes algorithm without updating, SEA data configuration, sequence size 100, threshold 1.263. Key: False alarm: ! / Miss: - / Hit: *.	131
Table 17.11: Two-class training input; Naive Bayes algorithm with updating, SEA data configuration, sequence size 100, threshold 1.227. Key: False alarm: ! / Miss: - / Hit: *.	132
Table 17.12: Two-class training input; Naive Bayes algorithm; SEA data configuration; no updating; sequence size 10; threshold 1.352.....	133
Table 17.13: Two-class training input; Naive Bayes algorithm; SEA data configuration; with updating; sequence size 10; threshold 1.342.....	134
Table 17.14: One-class training input; Naive Bayes algorithm without updating; 1v49 data configuration; sequence size 100; threshold 6.479.....	135

Table 17.15: One-class training input; Naive Bayes algorithm without updating; SEA data configuration; sequence size 100; threshold 6.479.....	136
Table 17.16: One-class training input; Naive Bayes algorithm without updating, 1v49 data configuration, sequence size 100, threshold 6.479.....	137
Table 17.17: One-class training input; Naive Bayes algorithm without updating, SEA data configuration, sequence size 100, threshold 6.479. Key: False alarm: ! / Miss: -/ Hit: *.....	138

Overview

Common wisdom in the cybersecurity community holds that over 80% of recorded intrusion cases are attributed to trusted insiders – people who have authorized access to systems and resources – and the threat is rising. Selected examples are: information/data thieves, moles, bank defrauders, and a long list of abusers who have access to the system and information resources of military sites, national laboratories, law enforcement agencies, tax authorities, banks, communications and power infrastructure, corporations, medical communities, motor-vehicle departments, and so forth.

Among the approaches for detecting insider abuse, profiling is the favored technology, and its preferred implementation mechanism, especially given a goal of detecting novel attacks or abuses, is anomaly detection. Anomaly detection, as a core technology, has progressed with disappointing slowness, the reasons for which are not difficult to ascertain: (1) The scientific principles of anomaly detection have not been well understood. (2) The sensor streams to which anomaly detectors are being applied (e.g., BSM and other audit data) have been acknowledged to be inadequate - it's not clear whether or not the right information is contained in these audit streams, at present, to support useful anomaly detection. (3) Even if the sensor streams were adequate, feature extraction and feature selection “filters” may be being applied to those sensor streams in ways that are not proved to be correct. Hence we are applying possibly incorrect filters to inadequate sensor streams, and feeding the untested results to anomaly detectors whose basic performance characteristics are little understood. It should come as no surprise, then, that anomaly detection has not advanced as quickly as might be hoped, and that profiling, which depends critically on anomaly detection, has not yet shown its promise. Various factors are largely responsible for the current situation, some of which are: insufficient scientific underpinnings for principled anomaly detection; inadequate tools and methodologies for assessing detector effectiveness; and lack of controlled benchmarks for calibrating anomaly-detector performance. As a consequence of these conditions, the intrusion-detection community is lost at sea without a compass.

The work reported here attended to what can be done to foment real progress in a time when intrusion-detection systems that truly work are in critical need. Four

areas were addressed:

- Science Base - Developing a basic science of anomaly detection and profiling, particularly to understand the intrinsic structure of categorical data sequences and the consequences of such structure on the nature of customized anomaly detectors and profilers. Principles developed here are used in realizing detection algorithms as well as in guiding generation of on-line evaluation testbeds.
- Algorithms - Developing a diverse suite of anomaly detectors for deployment, particularly as detection-system and profiling-system components, as well as for testbed applications.
- Benchmarks - Providing custom, calibrated testbeds for evaluating anomaly/profiling systems, using diverse kinds of user/system-type data (e.g., controlled, synthetic; natural, real-world; and hybrid) for replicated profiling and anomaly-based, contrastive experiments.
- Assessment - Providing statistically and methodologically rigorous assessment procedures for profiling and anomaly-detection effectiveness.

Highlights of this report include the following. Chapter 3 shows how an adversary, armed with knowledge of a detector's architecture, can craft an offensive mechanism that renders an anomaly-based intrusion detector blind to the presence of on-going, common attacks under particular detector/environment conditions. Chapter 4 identifies a previously unknown interaction between the characteristics of data and the architecture of a masquerade detection system. This interaction made it possible for a masquerader to avoid detection by interspersing his or her activities with so called never-before-seen commands. Chapters 6, 7 and 8 present program code, test data and performance analysis for the Congruent Sequence Detector (CSD) algorithm, which determines the abnormality of a sequence by how much it differs from the sequences found in the data that characterizes the normal behavior of the monitored subject. Chapters 10 and 11 present tools for creating benchmark datasets that correspond to particular characteristics. Chapters 13 and 14 present a series of benchmark datasets containing well-specified ground truth as well as data specifically tailored to address the insider-threat problem. Chapter 17 shows how a detector is assessed in the insider-threat environment, illustrating a careful and thorough analysis of an anomaly detector.

List of deliverables

This section lists the original project deliverables, each of which is hereby delivered by incorporation into this final report and/or its accompanying CD/DVD. Some of the original deliverables were de-scoped and omitted due to sponsor exigencies, and are so noted in their corresponding chapters.

1. **Descriptive characteristics of data.** This task was de-scoped and omitted due to sponsor exigencies.
2. **Impact of data characteristics on detection performance.** This task was de-scoped and omitted due to sponsor exigencies.
3. **Architectural aspects and impacts of anomaly-detection algorithms.** The interaction between the architectural aspects of a detection algorithm, such as detection mechanism and coverage, can result in unanticipated vulnerabilities that allow an adversary to undermine the detector.
4. **Interaction effects between data characteristics and detector architecture as applied to aspects of the insider-threat problem.** Detectors sometimes encounter problems due, not simply to a particular type of data, but to the conditions under which the data were encountered. Under other conditions, this interaction between the detector and the data would not be problematic.
5. **Review of facts acquired and lessons learned in developing a science base for anomaly detection.** This task was de-scoped and omitted due to sponsor exigencies.
6. **Program code and documentation for anomaly-detection algorithms.** Program code and documentation is presented for one implementation from one of the following classes of anomaly-detection algorithms: neural network detectors, probabilistic detectors, sequence time-delay embedding detectors, and frequentist detectors.
7. **Test data suites used in ascertaining the efficacy of the algorithms developed.** Test data are presented for use in ascertaining the efficacy of the CSD detection algorithm.
8. **Strengths and weaknesses of each detection algorithm implemented.** A selection of strengths and weaknesses of the CSD detection algorithm is presented, including detailed performance results.
9. **Tool for elucidating the sequential dependencies in categorical audit-type data.** Natural data have certain sequential dependencies that should be replicated in synthetic data. A tool is needed to ascertain the nature of such sequential dependencies.

10. **Tool for creating benchmark datasets that reflect the intrinsic structure which is characteristic of a given environment.** A given natural environment will yield data with intrinsic structure reflecting that environment. Replicating that structure requires a tool for incorporating that structure into synthetic benchmark data.
11. **Tool for generating benchmark datasets containing specified regularity.** Detectors can be sensitive to regularity in data. A tool is needed for generating a series of benchmark data sets that contains specific regularities spanning a region of sensitivity, thus creating an opportunity for performing a sensitivity analysis of a detector.
12. **Benchmark datasets from at least four natural data, monitored from real enterprise environments.** This task was de-scoped and omitted due to sponsor exigencies.
13. **Series of benchmark datasets containing well-specified, ground-truth structure, including training and test sets with known injected anomalies.** Accurate ground truth is a necessary and rare commodity in testing detection algorithms. Data with injected intrusive activity can be generated to include a table showing intrusion locations highly accurately.
14. **Benchmark dataset specifically tailored to address the insider-threat and profiling problems.** Most insider (masquerader) data are injected not with insider activity, but with activity from other users intended to imitate insiders. Because these data sets are unrealistic, data specific to the insider problem, specifically tailored to stress a detection algorithm, are needed.
15. **Data-gathering and generating methodologies.** This task was de-scoped and omitted due to sponsor exigencies.
16. **Guidance in the use of benchmark datasets for anomaly detection.** This task was de-scoped and omitted due to sponsor exigencies.
17. **Handbook of assessment procedures for validating detection algorithms and for assessing data environments, with particular emphasis on insider abuse.** This deliverable was changed to read: Assessing a detector in the insider-threat environment. The way a detector is evaluated can make a huge difference in how its performance is perceived. A detector should be assessed not only in terms of its success, but also its failures, as shown in an error analysis that reveals *why* the detector sometimes misperformed. This chapter illustrates a careful and thorough analysis of a masquerade detector.

Task 1

1. Descriptive characteristics of data

This task was de-scoped and omitted due to sponsor exigencies.

Task 2

2. Impact of data characteristics on detection performance

This task was de-scoped and omitted due to sponsor exigencies.

Task 3

3. Architectural aspects and impacts of anomaly-detection algorithms

Proem. The interaction between the architectural aspects of a detection algorithm, such as detection mechanism and coverage, can result in unanticipated vulnerabilities that allow an adversary to undermine the detector.

Over the past decade many anomaly-detection techniques have been proposed and/or deployed to provide early warnings of cyber-attacks, particularly of those attacks involving masqueraders and novel methods. To date, however, there appears to be no study which has identified a systematic method that could be used by an attacker to undermine an anomaly-based intrusion detection system. This chapter shows how an adversary can craft an offensive mechanism that renders an anomaly-based intrusion detector blind to the presence of on-going, common attacks. It presents a method that identifies the weaknesses of an anomaly-based intrusion detector, and shows how an attacker can manipulate common attacks to exploit those weaknesses. The chapter explores the implications of this threat, and suggests possible improvements for existing and future anomaly-based intrusion detection systems.

3.1 Introduction

In recent years, a vast arsenal of tools and techniques has been accumulated to address the problem of ensuring the availability, integrity and confidentiality of electronic information systems. Such arsenals, however, are frequently accompanied by equally vast “shadow” arsenals of tools and techniques aimed specifically at subverting the schemes that were designed to provide system security. Although a shadow arsenal can be viewed negatively as a formidable threat to the security of computer systems, it can also be viewed positively as a source of knowledge for identifying the weaknesses of current security tools and techniques in order to facilitate their improvement.

A small part of the security arsenal, and the focus of this work, is the anomaly-based intrusion-detection system. Anomaly-based intrusion-detection systems have

Table 3.1: Elements of methodology for undermining.

1.	Detection coverage (specifically, blind spots) of an anomaly detector.
2.	Where and how an attack manifests in sensor data.
3.	How to shift the manifestation from a covered spot to a blind one.

sought to protect electronic information systems from intrusions or attacks by attempting to detect deviations from the normal behavior of the monitored system. The underlying assumption is that such deviations may indicate that an intrusion or attack has occurred (or may still be occurring) on the system. Anomaly detection – detecting deviations from normal – is one of two fundamental approaches used in systems that seek to automate the detection of attacks or intrusions; the other approach is signature-based detection. Anomaly detection is typically credited with a greater potential for addressing security problems such as the detection of attempts to exploit new or unforeseen vulnerabilities (novel attacks), and the detection of abuse-of-privilege attacks, e.g., masquerading and insider misuse [3].

The promise of the anomaly-detection approach and its incorporation into a number of current automated intrusion-detection strategies (e.g., AT&T’s ComputerWatch, SRI’s Emerald, SecureNet, etc. [3]) underscores the importance of studying how attackers may fashion counter-responses aimed at undermining the effectiveness of anomaly-based intrusion-detection systems. Such studies are important for two reasons:

- to understand how to strengthen the anomaly-based intrusion-detection system by identifying its weaknesses; and
- to provide the necessary knowledge for guiding the design and implementation of a new generation of anomaly-based intrusion detectors that are not vulnerable to the weaknesses of their forebears.

This chapter lays out a method for undermining a well-known anomaly-based intrusion-detection system called stide [6], by first identifying the weaknesses of its anomaly-detection algorithm, and then by showing how an attacker can manipulate common attacks to exploit those weaknesses, effectively hiding the presence of those attacks from the detector’s purview. Stide was chosen primarily because it is freely available to other researchers via the Internet. Its accessibility not only encourages independent verification and replication of the work performed here, but it also builds on, and contributes to, a large body of previously published work that uses the stide detection mechanism.

To undermine an anomaly-based intrusion detector, an attacker needs to know the three elements described in Table 3.1. These elements set the framework for the chapter.

3.2 Approaches to undermining anomaly detectors

There are two approaches that would most obviously cause an anomaly detector to miss detecting the anomalous manifestation of an attack. The first of these two items describes the approach commonly found in the literature; the second describes the approach adopted by this study.

- modify the normal to look like the attack, i.e., incorporate the attack manifestations into the model of normal behavior; or
- modify the attack to make it appear as normal behavior.

In the intrusion detection literature, the most cited way to undermine an anomaly-based intrusion detection system is to incorporate undesired, intrusive behavior into the training data, thereby falsely representing “normal” behavior [3, 15, 20]. By including intrusive behavior explicitly into the training data, the anomaly detector is forced to incorporate the intrusive behavior into its internal model of normal and consequently lose the ability to flag future instances of that intrusive behavior as anomalous. Note that the anomaly detector is viewed as that component of an anomaly-based intrusion detection system solely responsible for detecting deviations from normal; it performs no diagnostic activities.

For example, one way to incorporate intrusive behavior into an anomaly detector’s model of normal behavior is to exploit the fact that behavior can change over time. Changing behavior requires the anomaly detector to undergo periodic on-line retraining. Should the information system undergo attacks during the retraining process, then the anomaly detector could inadvertently incorporate undesired attack behavior into its model of normal behavior [3]. The failure of an anomaly-based intrusion detector to detect intrusions or attacks can typically be attributed to contaminated training data, or to updating schemes that incorporate new normal behavior too quickly.

Undermining an anomaly-based intrusion detection system by simply incorporating intrusive behavior into its training data is too imprecise and abstract a method as to be practically useful to an attacker. Identifying and accessing the segment, feature or attribute of the data that will be used to train an anomaly detector, and then surreptitiously and slowly introducing the intrusive behavior into the training dataset, may require time, patience and system privileges that may not be available to an attacker. Moreover, such a scheme does not provide the attacker with any guarantees as to whether or not the act of subversion has been, or will be, successful. The incorporation of intrusive behavior into the training data is no guarantee that the anomaly detector will be completely blind to the attack when the attack is actually deployed. The attacker has no knowledge of how the anomaly detector perceives the attack, i.e., how the attack truly manifests in the data, and no knowledge concerning the conditions that may impede or boost the anomaly detector’s ability to detect the manifestation of the attack. For example, even if intrusive behavior were to be incorporated into an anomaly detector’s model of normal, it is possible that when the attack is actually deployed, it will interact with other conditions in the data environment (conditions that may not have been present during

the training phase), causing anomalous manifestations that *are* detectable by the anomaly detector. It is also possible for the anomalous manifestation of an attack to be detectable *only* when the detector uses particular parameter values. These points illustrate why it is necessary to determine precisely what kinds of anomalous events an anomaly detector may or may not be able to detect, as well as the conditions that enable it to do so.

These issues are addressed by determining the coverage of the anomaly detector in terms of anomalies (not in terms of attacks); this forms the basis of the approach. Only by knowing the kinds of anomalies that are or are not detectable by a given anomaly detector is it possible to modify attacks to manifest in ways that are not considered abnormal by a given anomaly detector. The coverage of an anomaly detector serves as a guide for an attacker to know precisely *how* to modify an attack so that it becomes invisible to the detector.

3.3 Detection coverage of an anomaly detector

Current evaluation techniques attempt to establish the detection coverage of an anomaly-based intrusion detection system with respect to its ability to detect attacks [42, 8, 9], but without establishing whether or not the anomalies detected by the system are attributable to the attack. Typically, claims that an anomaly-based intrusion detector is able to detect an attack are based on *assumptions* that the attack must have manifested in a given stream of data, that the manifestation was anomalous, and that the anomaly detector was able to detect that specific kind of anomaly.

The anomaly-based evaluation technique described in this section establishes the detection coverage of stide [6, 42] with respect to the types of anomalous manifestations that the detector is able to detect. The underlying assumption of this evaluation strategy is that no anomaly detection algorithm is perfect. Before it can be determined whether an anomaly-based intrusion detector is capable of detecting an attack, it must first be ascertained that the detector is able to detect the anomalous manifestation of the attack.

This section shows how detection coverage (in terms of a coverage map) can be established for stide. For the sake of completeness, and to facilitate a better understanding of the anomaly-based evaluation strategy, the stide anomaly-detection algorithm is described, followed by a description of the anomaly-based evaluation strategy used to establish stide’s detection coverage. A description and explanation of the results of the anomaly-based evaluation is given.

3.3.1 Brief description of the stide anomaly detector

Stide operates on fixed-length sequences of categorical data. It acquires a model of normal behavior by sliding a detector window of size DW over the training data, storing each DW -sized sequence in a “normal database” of sequences of size DW . The degree of similarity between test data and the model of normal behavior is based on observing how many DW -sized sequences from the test data are identical matches to any sequences from the normal database. The number of

mismatches between sequences from the test data and the normal database is noted. The anomaly signal, which is the detector’s response to the test data, involves a user-defined parameter known as the “locality frame” which determines the size of a temporally local region over which the number of mismatches is summed up. The number of mismatches occurring within a locality frame is referred to as the locality frame count, and is used to determine the extent to which the test data are anomalous. A detailed description of stide and its origins, can be found in [6, 42].

3.3.2 Evaluation strategy for anomaly detectors

It is not difficult to see that stide will only detect “unusual” or foreign sequences – sequences that do not exist in the normal database. Its similarity metric establishes whether or not a particular sequence exists in a normal database of sequences of the same size. Such a scheme means that any sequence that is foreign to the normal database would immediately be marked as an anomaly. However, this observation alone is not sufficient to explain the anomaly detector’s performance in the real world. There are two other significant issues that must be considered before the performance of the anomaly detector can be understood fully. Specifically:

- how foreign sequences actually manifest in categorical data; and
- how the interaction between the foreign sequences and the anomaly detection algorithm affects the overall performance of the anomaly detector.

In order to obtain a clear perspective of these two issues, a framework was established in [27, 40] that focused on the architecture and characteristics of anomalous sequences, e.g., foreign sequences. The framework defined the anomalous sequences that a sliding-window anomaly detector like stide would likely encounter, and provided a means to describe the structure of those anomalous sequences in terms of how they may be composed from other kinds of subsequences. The framework also provided a means to describe the interaction between the anomalous sequences and the sliding window of anomaly-detection algorithms like stide. Because the framework established how each anomalous sequence was constructed and composed, it was possible to evaluate the detection efficacy of anomaly detectors like stide on synthetic data with respect to examples of clearly defined anomalous sequences. The results of the evaluation showed the detection capabilities of stide with respect to the various foreign sequences that may manifest in categorical data, and how the interaction between the foreign sequences in categorical data and the anomaly-detection algorithm affected the overall performance of the anomaly detector.

3.3.3 Stide’s performance results

The most significant result provided by the anomaly-based evaluation of stide was that there were conditions that caused the detector to be completely blind to a particular kind of foreign sequence that was found to exist (in abundance) in real-world data [40]: a minimal foreign sequence. A minimal foreign sequence is foreign sequence whose proper subsequences all exist in the normal data. Put simply, a

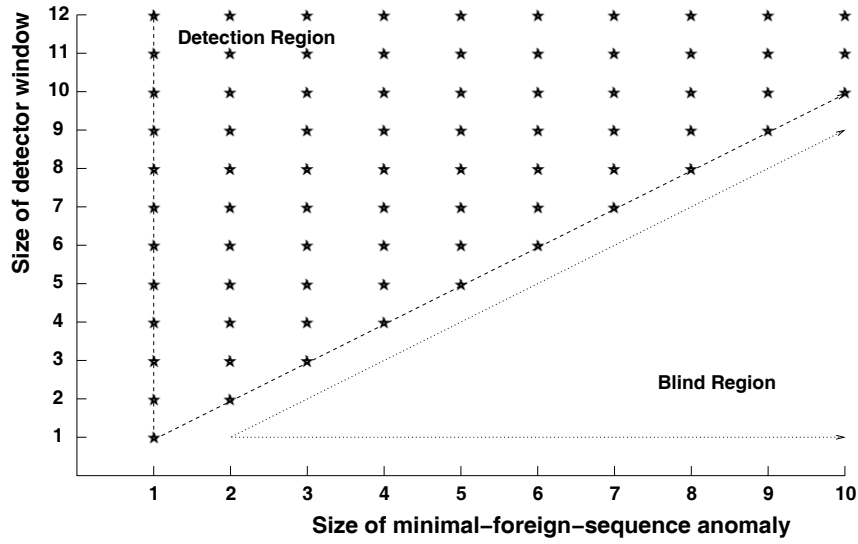


Figure 3.1: The detector coverage (detection map) for stide; A comparison of the size of the detector window (rows) with the ability to detect different sizes of minimal foreign sequence (columns). A star indicates detection.

minimal foreign sequence is a foreign sequence that contains within it no smaller foreign sequences.

For stide to detect a minimal foreign sequence, it is imperative that the size of the detector window is set to be equal to or larger than the size of the minimal foreign sequence. The consequence of this observation can be seen in Figure 3.1 which shows stide’s detection coverage with respect to the minimal foreign sequence. This coverage map for stide, although previously presented in [40], is shown again here as an aid to the reader’s intuition for the coverage map’s essential role in the subversion scheme.

The graph in the figure plots the size of the minimal foreign sequence on the x-axis and the size of the detector window on the y-axis. Each star marks the size of the detector window that successfully detected a minimal foreign sequence whose corresponding size is marked on the x-axis. The term *detect* for stide means that the minimal foreign sequence must have caused as at least one sequence mismatch. The diagonal line shows the relationship between the detector window size and the size of the minimal foreign sequence, a relationship that can be described by the function, $y = x$. The figure also shows a region of blindness in the detection capabilities of stide with respect to the minimal foreign sequence. This means that it is possible for a foreign sequence to exist in the data in such a way as to be completely invisible to stide. This weakness will presently be shown to be exploitable by an attacker.

3.4 Deploying exploits and sensors

At this point of the study, the first step in undermining an anomaly detector (see Table 3.1) has been completed; the detection coverage for stide has been established, and it was observed that the anomaly detector exhibited occasions of detection blindness with respect to the detection of minimal foreign sequences.

The following is a summary of the procedure that was performed in order to address the remaining two items in the method for subverting an anomaly detector listed in Table 3.1. The remaining two items are where and how an attack manifests in data, and how the manifestation of exploits can be modified to hide the presence of those exploits in the regions of blindness identified by the detection coverage for stide.

1. Install the sensor that provides the anomaly detector with the relevant type of data. In the present work, the sensor is the IMMSEC kernel patch for the Linux 2.2 kernel [38]. The kernel patch records to a file the system calls made by a pre-determined set of processes.
2. Download the `passwd` and `traceroute` exploits and determine the corresponding system programs that these exploits misuse.
3. Execute the system program under normal conditions to obtain a record of normal usage, to obtain normal data. An account of what is considered normal conditions and normal usage of the system programs that correspond to both exploits is described in section 3.5.2.
4. Deploy the exploits against the host system to obtain the data recording the occurrence of the attacks.
5. Identify the precise manifestation of the attacks in the sensor data.
6. Using the normal data obtained from step 3, and the intrusive data obtained from step 4, deploy stide to determine if the anomaly detector is capable of detecting the unmodified exploits that were simply downloaded, compiled and executed. This is performed in order to establish the effectiveness of the subversion process. If stide is able to detect the unmodified exploits but not the modified exploits, then the subversion procedure has been effective.
7. Using information concerning the kind of events that stide is blind to, modify the attacks and show that it is possible to make attacks that were once detectable by stide, undetectable for detector window sizes one through six.

3.5 Where and how an attack manifests in the data

This section addresses the second item in the list of requirements for undermining an anomaly detector – establishing where and how an attack manifests in sensor data (see Table 3.1) – by selecting two common exploits, deploying them, and establishing how and where they manifest in the sensor data. Steps 2 to 5 of the method laid out above are covered by this section.

3.5.1 Description and rationale for the exploits chosen

The attacks selected for this study are examples of those that stide is designed to detect, i.e., attacks that exploit privileged UNIX system programs. UNIX system programs typically run with elevated privileges in order to perform tasks that require the authority of the system administrator – privileges that ordinary users are not typically afforded. The authors of stide have predominantly applied the detector towards the detection of abnormal behavior in such privileged system programs, because exploiting vulnerabilities to misuse privileged system programs can potentially bestow those extra privileges on an attacker [10].

Two attacks were chosen arbitrarily out of several that fulfill the requirement of exploiting UNIX system programs. The two attacks chosen will be referred to as the `passwd` and `traceroute` exploits. The `passwd` exploit takes advantage of a race condition between the Linux kernel and the `passwd` system program; the `traceroute` exploit takes advantage of a vulnerability in the `traceroute` system program.

`passwd` is a system program used to change a user’s password [7]. The program allows an ordinary user to provide his or her current password, along with a new password. It then updates a system-wide database of the user’s information so that the database contains the new password. The system-wide database is commonly referred to as the `/etc/passwd` or the `/etc/shadow` file. A user does not normally have permission to edit this file, so `passwd` must run with root privileges in order to modify that file. The exploit that misuses the `passwd` system program does so by employing a race condition that is present in the Linux kernel to debug privileged processes.

Normally, the `passwd` system process performs only a restricted set of actions that consists of editing the `/etc/passwd` and/or the `/etc/shadow` file. However, the `passwd` system process can be made to do more, because of this race condition. Using an unprivileged process, an attacker can alter or “debug” the `passwd` system process and force it to execute a command shell, granting the attacker elevated privileges.¹ Details of race conditions in the Linux kernel are given in [31] and [37]. The `passwd` exploit was obtained from [33].

The `traceroute` network diagnostic utility is a system program that is usually employed by normal users to gather information about the availability and latency of the network between two hosts [11]. To accomplish this task, the system program `traceroute` must have unrestricted access to the network interface, a resource provided only to privileged system programs. However, a logic error in the `traceroute` system program allows an attacker to corrupt the memory of the process by specifying multiple network gateways on the command line [36]. The `traceroute` exploit uses this memory corruption to redirect the process to instructions that execute a command shell with the elevated privileges of the `traceroute` system program [13]. More detail on this memory corruption vulnerability is provided in [36]. The `traceroute` exploit was obtained from [13].

Several key features make certain attacks or exploits likely candidates for sub-

¹In industry parlance, the instructions injected by the exploit are termed “shellcode”, and the shell in which an intruder gains elevated privileges is a “rootshell.”

verting sequence-based anomaly detectors such as stide. The subversion technique presented in this chapter is more likely to be effective when:

- the vulnerability exploited involves a system program that runs with elevated (root) privileges;
- the vulnerability allows an attacker to take control of the execution of the system program, giving the attacker the ability to choose the system kernel calls or instructions that are issued by the system program;
- the attack does not cause the system program to behave anomalously (e.g. produce an error message in response to an invalid input supplied by the attacker) before the attack/attacker can take control of the execution of the system program;
- the system kernel calls occurring after the “point of seizure”, i.e., the point in the data stream at which the attacker first takes control of the system program, include any or all of `execve`, or `open/write`, or `chmod`, or `chown`, or any other system kernel call that the attacker can use to effect the attack.

3.5.2 Choice of normal data

It is unreasonable to expect an attacker to be able to identify and access the precise segment, feature or attribute of the data that can be used to train an anomaly detector. The time, patience and system privileges required to do so may simply not be available to an attacker. However, since training data is vital to the function of an anomaly detector, the attacker has to construct an approximation of the training data that may have been used by the anomaly detector if he or she desires to exploit a blind spot of the detector.

For anomaly detectors like stide, i.e., anomaly detectors that monitor system programs, training data can be approximated more easily, because system programs typically behave in very set and regimented ways. For example, the `passwd` and `traceroute` system programs are limited in the number of ways that they can be used, and as a result it is possible to make reasonable assumptions about how these programs would be regularly invoked.

These assumptions may be aided by the wealth of easily accessible documentation that typically accompanies each system program, as well as by any general knowledge or experience already acquired by the attacker. It is important to note, however, that the success of this method for undermining stide relies on the attacker’s being able to approximate normal usage of the system program.

To be successful at undermining stide, the attacker does not need to obtain every possible example of a system program’s normal behavior. If the anomalous manifestation of an exploit can already be crafted by an extremely reduced subset of normal behavior, then it can only be expected that more examples of normal behavior contribute to an increased number of ways with which to construct the anomalous manifestation of the exploit.

For the `passwd` system program, normal data was obtained by executing the `passwd` system program with no arguments, and then by following the instructions displayed by the program to input the user's current password once, and then their new password twice. In other words `passwd` was invoked to expire an old password and install a new one.

For the `traceroute` system program, normal data was obtained by executing `traceroute` to acquire diagnostic information regarding the network connectivity between the local host and the Internet site `nis.nsf.net`. This site was chosen because it is the simplest example of using `traceroute`, based on the documentation provided with the program itself [11].

3.5.3 Establishing attack manifestations in sensor data

Two issues are addressed in this subsection. The first is whether the attacks embodied by the execution of the chosen exploits actually manifested in the sensor data, and the second is whether the manifestation is an anomalous event detectable by `stide`. Simply because an attack can be shown to manifest in the sensor data does not necessarily mean that the manifestation is automatically anomalous. It is necessary to establish that the manifestation of the exploits are initially detectable by `stide` in order to show that any modifications to the same exploits effectively render them undetectable by the same detector.

Before proceeding any further it is necessary to define what is meant by the term *manifestation* within the scope of this study. The manifestation of an attack is defined to be that sequence of system calls issued by the exploited/privileged system program, and due to the presence and activity of the exploit. The remainder of this section describes how the manifestations of the two exploits, `passwd` and `traceroute`, were obtained.

`passwd`.

The `passwd` exploit was downloaded from [33]; then it was compiled and deployed. There were no parameters that needed to be set in order to execute the exploit. The successful execution of the exploit was confirmed by checking that elevated privileges were indeed conferred.

The manifestation of the `passwd` exploit was determined manually. An inspection of the source code for both the `passwd` exploit and that portion of the Linux kernel responsible for the race condition vulnerability identified the precise system calls that were attributable to the attack. The sequence of system calls that comprise the manifestation of the attack embodied by the `passwd` exploit is `setuid`, `setgid`, `execve`. `Stide` was then deployed, using the normal data described above as training data, plus the test data comprised of the data collected while the `passwd` exploit was executed. `Stide` was run with detector window sizes ranging from 1 to 15. It was found that the attack was detectable at all detector window sizes. More precisely, the attack was detectable by `stide` because `setuid`, `setgid`, and `execve` were all foreign symbols. From the detection map of `stide` in Figure 3.1, it can be seen that `stide` is capable of detecting size-1

foreign symbols at any detector window size.

traceroute.

The `traceroute` exploit was downloaded from [13]; then it was compiled and deployed. The `traceroute` exploit expects values for two arguments. The first argument identifies the local platform, and the second argument is a hexadecimal number that represents the address of a specific function in memory. This address is overwritten to point to an attacker-specified function. The successful execution of the exploit was confirmed by checking that elevated privileges were indeed conferred.

The manifestation of the `traceroute` exploit was determined manually. An inspection of the source code for the `traceroute` exploit as well as for the `traceroute` system program, identified the precise system calls that were attributable to the attack. The sequence of system calls that comprise the manifestation of the attack embodied by the `traceroute` exploit is: `brk`, `brk`, `brk`, `setuid`, `setgid`, `execve`. Mirroring the deployment strategy for `passwd`, `stide` was trained on the previously collected `traceroute` normal data, and run with detector-window sizes 1-15. The attack was shown to be detectable at all window sizes, because `setuid`, `setgid`, and `execve` were all foreign symbols.

3.6 Manipulating the manifestation; modifying exploits

Three vital items of knowledge have been established up to this point: the characteristics of the minimal-foreign-sequence event that `stide` is sometimes unable to detect; the conditions ensuring that `stide` does not detect such an event (the detector-window size must be smaller than the size of the minimal foreign sequence); and the fact that `stide` is completely capable of detecting the two chosen exploits when they were simply executed on the host system without any modifications. This means that the anomaly detector is completely effective at detecting these exploits should an attacker decide to deploy them.

How can these exploits be modified so that the anomaly detector does not sound the alarm when the modified exploits are deployed? How can an attacker provide his or her attack(s) with every opportunity to complete successfully and stealthily? This section shows how both exploits, guided by the detection map established for `stide`, can be modified to produce manifestations (or signatures) in the sensor data that are not visible to the detector.

3.6.1 Modifying `passwd` and `traceroute`

In aiming to replace the detectable anomalous manifestations of the exploits with manifestations that are undetectable by `stide`, there are two points that must be considered. Recall that each exploit embodies some goal to be attained by the attacker, e.g., elevation of privileges.

First, because the method for achieving the goal that is embodied by each exploit, `passwd` and `traceroute`, produces an anomalous event detectable by

Table 3.2: The system calls that implement each of three methods that attempt to achieve the same goal of securing an interactive root account accessible to the attacker.

	Description of method	System calls that implement method
1	Changing the access rights to the <code>/etc/passwd</code> file in order to give the attacker permission to modify the file (write permission)	<code>chmod, exit</code>
2	Changing the <i>ownership</i> of the <code>/etc/passwd</code> file to the attacker	<code>chown, exit</code>
3	Opening the <code>/etc/passwd</code> file to append a new user with root privileges	<code>open, write, close, exit</code>

stide, namely a foreign symbol, another method for achieving the same goal must be found to replace it. Note that the goal of both exploits is the typical one of securing an interactive shell with elevated privileges. Interestingly, the new means of achieving the same goal involves changing the value of only one variable in both exploit programs.

Second, the new method of achieving the same goal must not produce any manifestation that is detectable by stide. Although this could mean that both exploits are modified so that their manifestations appear normal, i.e., their manifestations match sequences that already exist in the normal data, it is typically more difficult to do this than to cause the exploits to manifest as foreign sequences. The difficulty lies in the fact that the kinds of normal sequences that can be used to effect an attack may be small. This makes it more likely that an attacker may require sequences that lie outside the normal vocabulary, i.e., foreign sequences.

3.6.2 New means to achieve original goals in exploit programs

In Section 3.5.3 it was shown that the execution of the `passwd` and `traceroute` exploits were detectable by stide because both exploits manifested anomalously as the foreign symbols `setuid`, `setgid`, and `execve`. Any attack that introduces a foreign symbol into the sensor data that is monitored by stide, will be detected. This is because foreign symbol manifestations lie in the visible region of stide's detection map. In order for the `traceroute` or `passwd` exploits to become undetectable by stide, they must not produce the system calls `setuid`, `setgid`, and `execve`. Instead an alternate method causing the exploits to manifest as minimal foreign sequences is required. Only system calls that are already present in the normal data can be the manifestation of the exploits.

For the `passwd` exploit, another method for achieving the same goal of securing an interactive shell with elevated privileges that does not involve the foreign

symbols `setuid`, `setgid`, and `execve` would be to cause the exploit program to give the attacker permission to *modify* the `/etc/passwd` file. With such access, the attacker can then edit the accounts and give him or herself administrative privileges, to be activated upon his or her next login. The system calls required to implement this method are `chmod` and `exit`. These two calls are found in the normal data for the `passwd` system program.

There are at least two other methods that will achieve the same goal. A second method would be to give the attacker *ownership* of the `/etc/passwd` file, and a third method would be to make the affected system program directly edit the `/etc/passwd` file to add a new administrative (root) account that is accessible to the attacker. The system calls that would implement all three methods respectively are listed in Table 3.2.

For the `traceroute` exploit, the other method for achieving the same goal of securing an interactive shell with elevated privileges that does not involve the foreign symbols `setuid`, `setgid`, and `execve`, is to make the affected system program directly edit the `/etc/passwd` file to add a new administrative (root) account that is accessible to the attacker. The system calls required to implement this method are `open`, `write`, `close`, and `exit`. All these system calls can be found in the normal data for the `traceroute` system program.

3.6.3 Making the exploits manifest as minimal foreign sequences

In the previous subsection, the two exploits were made to manifest as system calls that can be found in the normal data for the corresponding `passwd` and `traceroute` system programs. This is still insufficient to hide the manifestations of the exploits from `stide`, because even though system calls that already exist in the normal data were used to construct the new manifestation of each exploit, the order of the system calls with respect to each other can still be foreign to the order of system calls that typically occur in the normal data. For example, even if `chmod` and `exit` both appear in the `passwd` normal data, both calls never appear sequentially. This means that the sequence `chmod`, `exit`, is a foreign sequence of size 2, foreign to the normal data. More precisely, this is a minimal foreign sequence of size 2, because the sequence does not contain within it any smaller foreign sequences or foreign symbols.

As a consequence, `stide` with a detector window of size 2 or larger would be fully capable of detecting such a manifestation. In order to make the manifestation invisible to `stide`, it is necessary to increase the size of the minimal foreign sequence. Increasing the size raises the chances of falling into `stide`'s blind spot. Referring to Figure 3.1, it can be seen that the larger the size of the minimal foreign sequence, the larger the size of the blind spot.

To increase the size of the minimal foreign sequence, the short minimal foreign sequences that are the manifestations of both exploits (`chmod`, `exit` for the `passwd` exploit, and `open`, `write`, `close`, and `exit` for the `traceroute` exploit) must be padded with system calls from the normal data that would result in larger minimal foreign sequences with common subsequences. For example, for

`passwd` the short minimal foreign sequence that is the manifestation of the new method described in the previous section is `chmod`, `exit`. This is a minimal foreign sequence of size 2. To increase this minimal foreign sequence it can be seen that in the normal data for `passwd`, the system call `chmod` is followed by the sequence `utime`, `close`, `munmap`, and elsewhere in the normal data, `munmap` is followed by `exit`. These two sequences

1. `chmod`, `utime`, `close`, `munmap`
2. `munmap`, `exit`
can be concatenated to create a third sequence
3. `chmod`, `utime`, `close`, `munmap`, `exit`.

A method of attack can be developed which manifests as this concatenated sequence. This method is functionally equivalent to the method developed in the previous subsection; it gives the attacker permission to modify `/etc/passwd` with the `chmod` system call and exits with the `exit` system call. The three system calls `utime`, `close`, and `munmap` are made in such a way that they do not alter the state of the system.

If `stide` employed a detector window of size 2, and the detector window slid over the manifestation of the exploit that is the sequence `chmod`, `utime`, `close`, `munmap`, `exit`, no anomalies would result; no alarms would be generated because the manifestation no longer contains any foreign sequences of size 2. However, if `stide` employed a detector window of size 3, a single anomaly would be detected, namely the minimal foreign sequence of size 3, `close`, `munmap`, `exit`, which would result in an alarm.

The simple example given above describes the general process for creating the larger minimal foreign sequences required to fool `stide`. By performing an automated search of the normal data it is possible to find all sequences that can be used by an attacker as padding for the manifestation of a particular exploit. The general process for creating larger minimal foreign sequences was automated and used to modify both the `passwd` and `traceroute` exploits.

It is important to note that because `stide` only analyzes system calls and not their arguments, it is possible to introduce system calls to increase the size of minimal foreign sequences without affecting the state of the system. Executing system calls introduced by the attacker that are aimed at exploiting `stide`'s blind spot need not cause any unintended side-effects on the system because the arguments for each system call is ignored. It is therefore possible to introduce system calls that do nothing, such as reading and writing to an empty file descriptor, or opening a file that cannot exist. This point argues for using more diverse data streams in order to provide more effective intrusion detection. Analyzing only the system call stream may be a vulnerability in anomaly detectors.

3.7 Evaluating the effectiveness of exploit modifications

A small experiment is performed to show that the modified exploits were indeed capable of fooling stide. As shown in the previous section, a single deployment of a modified exploit is accompanied by a parameter that determines the size of the minimal foreign sequence that will be the manifestation of the exploit. Each exploit was deployed with parameter values that ranged between 2 and 7. A minimum value of 2 was chosen, because it is the smallest size possible for a minimal foreign sequence. The maximum value chosen was 7, because a minimal foreign sequence of size 7 would be invisible to stide employing a detector window of size 6. In the literature, stide is often used with a detector window of size 6 [10, 42]. 6 has been referred to as the “magic” number that has caused stide to begin detecting anomalies in intrusive data [10, 23]. Using a detector window of size 6 in this experiment serves to illustrate a case where 6 may not be the best size to use because it will miss detecting exploits that manifest as minimal foreign sequences of size 7 and higher.

Each of the two exploits were deployed 6 times, one for each minimal foreign sequence size from 2 to 7. For each execution of an exploit, stide was deployed with detector window sizes 1 to 15. 1 was chosen as the minimum value simply because it is the smallest detector window size that the detector can be deployed with, and 15 was chosen as the maximum arbitrarily.

3.7.1 Results

The x-axis for the graph in Figure 3.2 represents the size of the minimal foreign sequence anomaly, and the y-axis represents the size of the detector window. Each star marks the size of the detector window that successfully detected a minimal foreign sequence whose corresponding size is marked on the x-axis. The term detect for stide means that the manifestation of the exploit must have registered as at least one sequence mismatch. Only the results for `traceroute` are presented. The results for `passwd` are very similar and have been omitted due to space limitations.

The graph in Figure 3.2 mirrors the detection map for stide, showing that the larger the minimal foreign sequence that is the manifestation of an exploit, the larger the detector window required to detect that exploit. Each circle marks the intersection between the size of the minimal foreign sequence that is the manifestation of the exploit and the size of the detector window used by stide, namely 6. Within each circle the presence of the star indicates that the manifestation of the exploit was detected by stide with a window size of 6.

Each successive circle along the x-axis at $y = 6$ depicts a shift in the manifestation of the exploit in terms of the increasing size of the minimal foreign sequence. These shifts are due to having modified the exploit. The arrows indicate a succession of modifications. For example, without any modification the exploit will naturally manifest as a foreign symbol in the data stream; this is represented by the circle at $x = 1, y = 6$. The first modification of the exploit resulted in a minimal foreign sequence of size 2; this is represented by the circle at $x = 2, y = 6$ pointed to by the arrow from the circle at $x = 1, y = 6$. The second modification yields

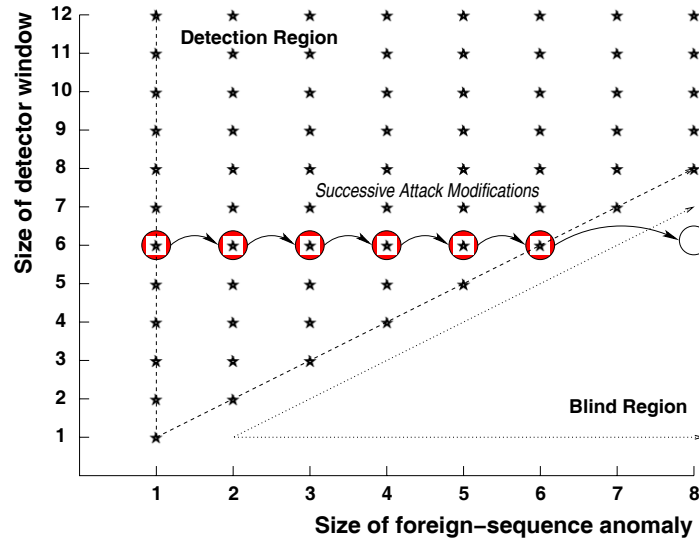


Figure 3.2: The manifestation of each version of the `traceroute` exploit plotted on the detector coverage map for stide, assuming that stide has been configured with a detector window size of 6. Each version of the exploit can be detected by fewer and fewer configurations of stide until the last is invisible.

a size-3 foreign sequence, and so forth. There is no circle at $x = 7$ because it was impossible to modify the exploit to shift its manifestation to a size-7 minimal foreign sequence, given the normal data for `traceroute`.

To summarize, if stide were deployed with a detector window of size 6, then it is possible to modify the `traceroute` exploit incrementally, so that it manifests as successively larger minimal foreign sequences, until a size is reached (size 7) at which the manifestation falls out of stide’s visible detection range, and into its blind spot. This shows that it is possible to exert control over a common exploit so that its manifestation is moved from an anomaly detector’s detection region, to its region of complete blindness. Such movement of an exploit’s manifestation effectively hides the exploit from the detector’s view.

3.8 Discussion

The results show that it is possible to hide the presence of common exploits, such as `passwd` and `traceroute`, from stide by modifying those exploits so that they manifest only within stide’s detection blind spot. Achieving an attack’s objectives is not affected by the modifications to the exploit programs; neither is the training data tampered with in order to render an anomaly detector blind to the attacks. Note that, at present, these results can be said to be relevant only to other anomaly-based intrusion detection systems that employ anomaly detectors operating on sequences of categorical data. Although the results make no claims about other families of

anomaly detectors that, for example, employ probabilistic concepts, it is possible that the methods described in this study may be applicable to a broader range of anomaly detectors.

The results presented in this chapter show that it is also possible to control the manifestation of an attack so that the manifestation moves from an area of detection blindness to an area of detection clarity for stide. Figure 3.2 shows the results of modifying the manifestation of an exploit in controlled amounts until the manifestation falls outside the anomaly detector's detection range.

3.8.1 Implications for anomaly detector development

By identifying the precise event and conditions that characterize the detection blindness for stide and showing that real-world exploits can be modified to take advantage of such weaknesses, one is forewarned not only that such weaknesses exist, but also that they present a possible and tangible threat to the protected system. It is now possible to mitigate this threat by, for example, combining stide with another detector that could compensate for the problems inherent in the stide detection algorithm. The variable sequence size model explored by Marceau [23] seems to be a promising step toward addressing the weakness in stide. Because detection coverage has been defined in a way that is pertinent to anomaly detectors, i.e., in terms of the kinds of anomalies that can be detected by a given anomaly detector rather than in terms of intrusions, it is also possible to compose anomaly detectors to effect full coverage over the detection space.

It is interesting to note that the ease with which an attacker can introduce sequences into the system call data suggests that sequences of system calls may not be a sufficiently expressive form of data to allow an anomaly detector to more effectively monitor and defend an information system. Increasing the number of different kinds of data analyzed, or changing the kind of data analyzed by an anomaly detector, may make an impact on the effectiveness of the intrusion-detection capabilities of an anomaly detector.

3.8.2 Implications for anomaly detector evaluation

There are a few benefits of an anomaly-based evaluation method, an evaluation method focused on how well anomaly detectors detect anomalies. First, the results of an anomaly-based evaluation increases the scope of the results. In other words, what was established to be true with respect to the anomaly-detection performance of the anomaly detector on synthetic data will also be true of the anomaly detector on real-world data sets. This cannot be said of current evaluation procedures for anomaly detectors because current evaluation schemes evaluate an anomaly detector in terms of how well it detects attacks. This constrains the scope of the results to the data set used in the evaluation because an attack that manifests as a specific kind of anomaly in one data set may no longer do so in another data set due to changing normal behavior.

Second, the results of an anomaly-based evaluation can only contribute to increasing the accuracy of anomaly detectors performing intrusion detection. Cur-

rent evaluation methods do not establish the detection capabilities of an anomaly detector with regard to the detection of the anomalous manifestations of attacks. The fact that attacks may manifest as different types of anomalies also means that different types of anomaly detectors may be required to detect them. If anomaly detectors are not evaluated on how well they detect anomalies, it is difficult to determine which anomaly detector would best suit the task of detecting a given type of attack. The events that anomaly detectors directly detect are anomalies, but typically anomaly detectors are evaluated on how well they detect attacks, the events that anomaly detectors do not detect except by making the assumption that attacks manifests as those anomalies detected by an anomaly detector.

3.9 Related work

The concept of modifying an attack so that it successfully accomplishes its goal while eluding detection is not a new one. Ptacek and Newsham [32] highlighted a number of weaknesses that the network intrusion detection community needed to address if they were to defeat a wily attacker using network signature-based intrusion detection systems. Although the work presented in this chapter differs in that it focuses on anomaly-based intrusion detection, it strongly reiterates the concern that an unknown weakness in an intrusion detection system creates a “dangerously false sense of security.” In the case of this work, it was shown that a weakness in an anomaly detector could realistically be exploited to compromise the system being protected by that detector.

Wagner and Dean [41] introduced a new class of attacks against intrusion detection systems which they called the “mimicry attack”. A “mimicry attack” is where an attacker is able to “develop malicious exploit code that mimics the operation of the application, staying within the confines of the model and thereby evading detection ...” Wagner and Dean studied this class of attack theoretically, but until this present chapter, no study has shown if it were possible to create and deploy a mimicry attack in the real-world that truly affected the performance of an intrusion-detection system.

The present study confirms that the class of mimicry attacks does pose a serious threat to an anomaly-based intrusion detection system. By modifying common real-world exploits to create examples of this class of attack, this study also shows how mimicry attacks are able to undermine the protection offered by an anomaly-based intrusion detection system.

3.10 Conclusion

This study has shown how an anomaly-based intrusion detection system can be effectively undermined by modifying common real-world exploits. It presented a method that identified weaknesses in the architecture of an anomaly-based intrusion detector, and showed how an attacker can effectively modify common exploits to take advantage of those weaknesses in order to craft an offensive mechanism

that renders an anomaly-based intrusion detector blind to the on-going presence of those attacks.

The results show that it is possible to hide the presence of the `passwd` and `traceroute` common exploits from stide by modifying those exploits so that they manifest only within stide's detection blind spot. The results also show that it is possible to control the manifestation of an attack such that the manifestation moves from an area of detection clarity to one of detection blindness for stide.

Task 4

Interaction effects: data characteristics vs. detector architecture

Proem. Detectors sometimes encounter problems due, not simply to a particular type of data, but to the conditions under which the data were encountered. Under other conditions, this interaction between the detector and the data would not be problematic.

4.1 Introduction

Within the domain of insider-threat is the significant problem of masquerader detection, i.e., discriminating between a legitimate user on a computer system and a masquerader impersonating that user. To address this problem, we have employed the naive Bayes machine learning algorithm. The *training* portion of the naive Bayes algorithm is used to build a model of “self” and “nonself” for each legitimate user. To build these models, the algorithm uses data collected both while monitoring the particular legitimate user and also while monitoring other users. Respectively, these are called the *self* and *nonself training data* and the models built are the *self* and *nonself models*. Once these models are built, the *test* portion of the algorithm can be used to estimate the probability that a user of the system actually is the user he or she purports to be. If this probability is high, then the user is believed to be legitimate. However, if the probability is low, then the user is viewed as suspicious, and possibly indicative of masquerader activity.

We have designed and implemented this masquerade detector based on naive Bayes. It has performed well in all our evaluations up to this point. It has been used to build profiles of users based on the commands they type at the UNIX command line, and then to distinguish between the data of the expected user and others [28]. On an evaluation dataset of this nature, the detector has performed with a lower

cost of operation than any other comparable detector on the same data.

It is within this context that we began our investigation into the interaction between data characteristics and detector architecture. While the naive Bayes algorithm has performed well up to this point, we have already begun to see that the performance of the detector varies depending on the environment. In one particularly significant case, small changes to the environment have had a major effect on detector performance. We were investigating the effect of *enriching* the command-line data. Normally, the command-line typed by the user is truncated so that only the command itself is fed to naive Bayes. In this experiment, we preserved other features of the command line such as shell grammar, alias information, and flags [25]. With such enriched data, we obtained a 16% improvement in the detection rate while only increasing the false alarm rate from 4.7% to 5.7%. In both the experiments with the original and the enriched data, there were instances where a masquerader went undetected. However, we noticed that in the enriched data, while more masqueraders were detected, some of those who were not detected *were undetectable no matter which user they were impersonating*.

While overall performance improved, the data enrichment also gave rise to these undetectable *super masqueraders*. Since no change was made to the detector, and the only change to the data was enrichment, it seemed that enriching the data altered some characteristic to make super-masquerading feasible. We observed that with enrichment came an increased alphabet size, and an increased diversity in the types of data that are fed into naive Bayes. We examined the types of commands that the super masqueraders used when they were successfully impersonating other users, and we noticed a prevalence of command-lines that had never been used before, by the legitimate user or any other user. Could these never-before-seen commands assist a masquerader in remaining undetected? What interaction between these commands and the naive Bayes algorithm would explain this effect?

4.2 Hypothesis and approach

We propose the following hypothesis. *Never-before-seen commands (NBSCs) tend to lower the naive Bayes anomaly score such that it drops below the threshold anomaly score, causing decisions to favor the user instead of the masquerader, and therefore resulting in missed alarms.* We will investigate this hypothesis in three stages.

- First, we will examine the mathematical underpinnings of the naive Bayes algorithm in order to find a theoretical justification for the observed effect of NBSCs.
- Second, we will empirically measure the effects of NBSCs carefully controlling for confounding factors in the environment.
- Third, after compiling strong evidence to support the hypothesis, we will examine what can be done to mitigate the adverse effect of these never-before-seen commands.

In the next three sections, we will describe each of these stages and what we found during our investigation. As we investigate this hypothesis, we will gather evidence as to how data characteristics – e.g., presence and number of NBSCs – interact with the detector architecture – e.g., employment of the naive Bayes algorithm – to affect detector performance. We will also examine how adverse effects on performance can be mitigated once they are identified, thereby increasing the efficacy of a masquerade detector.

4.3 Examining the mathematical model of naive Bayes

Why should NBSCs have the hypothesized effect on naive Bayes? In this section, we will answer this question. We will formulate a mathematical model of the calculation performed by naive Bayes when it estimates the probabilities that a block of commands is legitimate or suspicious. Fundamentally, this mathematical model consists of these two probability calculations, called the *self* and *nonself* probability estimates. These calculations form the core of the masquerade detection architecture because it is in their comparison that our detector determines whether a block is legitimate or suspicious. We will show, through algebraic analysis of these calculations that NBSCs tend to increase the self probability with respect to the nonself probability. Hence, the detector tends to decide that blocks containing NBSCs are legitimate.

During its training phase, naive Bayes simply counts the number of times each command appears in the self training data and the nonself training data. It also keeps a running tally of the total number of commands seen in each of these two sets of data. After naive Bayes has processed the training data, it enters the test phase. During this phase, a block of commands is fed to naive Bayes and it computes an anomaly score for the block. We will now explain in more detail how this score is calculated.

4.3.1 Findings

We let b be the size of the block of commands fed to naive Bayes. We will refer to the individual commands in this block as c_1, c_2, \dots, c_b . The primary purpose of naive Bayes is to attribute this block of commands either to *self* (i.e., the legitimate user) or to *nonself* (a masquerader). We let s_t and n_t be the total number of commands that naive Bayes processed during its training on the self training data and the nonself training data respectively. We let $s[c_i]$ be the number of times the command c_i appeared in the self training data and $n[c_i]$ be the number of times c_i appeared in the nonself training data.

Now, a very simplistic way to estimate the probability that a block of commands came from the legitimate user is shown in Eqn 4.1.

$$P_1(S) = \prod_{i=1}^b \frac{s[c_i]}{s_t} \quad (4.1)$$

Each factor in the product is the probability that a particular command in the block would have been typed by the legitimate user, given the training data. The overall

product is the probability that each of these commands would be typed independently by the legitimate user. Similarly, the probability that the block of commands was typed by another user can be calculated based on the nonself training data, as shown in Eqn 4.2.

$$P_1(N) = \prod_{i=1}^b \frac{n[c_i]}{s_t} \quad (4.2)$$

These very simple calculations, based solely on the probabilities as observed in the training data, are quite similar to the probability estimates that naive Bayes calculates. However, to make them exact, they must be further refined.

First, note that each probability is a number between 0 and 1. When we start multiplying these probabilities together, the product gets very small, very fast. To prevent rounding errors in this calculation, it is possible to perform the same calculations on the logarithmic scale, as shown in Eqn 4.3.

$$\begin{aligned} \log P_2(S) &= \sum_{i=1}^b (\log(s[c_i]) - \log(s_t)) \\ \log P_2(N) &= \sum_{i=1}^b (\log(n[c_i]) - \log(n_t)) \end{aligned} \quad (4.3)$$

Next, suppose that a command c_i does not appear in the self training data. Then, this sum will involve the logarithm of zero which will take the whole sum negative to infinity. To prevent one command from exerting such total control over the decision procedure, it is standard to assume that every command has some small but nonzero prior probability of occurring. We call this prior the pseudocount and denote it p . In the probability calculations, it is handled as though p additional examples of each command were added to the self and nonself training data. Since this pseudocount is added for each command, we must also factor in the increase in the total number of unique commands in the self and nonself training data. To do so, we will need to represent the total number of commands that are possible (often called the alphabet size). We let α represent the alphabet size. Eqn 4.4 shows the probability calculations with the pseudocount:

$$\begin{aligned} \log P_3(S) &= \sum_{i=0}^b (\log(s[c_i] + p) - \log(s_t + \alpha p)) \\ \log P_3(N) &= \sum_{i=0}^b (\log(n[c_i] + p) - \log(n_t + \alpha p)) \end{aligned} \quad (4.4)$$

Now, these two probability calculations – $P_3(S)$ to estimate the probability of self and $P_3(N)$ to estimate the probability of nonself – are exactly the calculations performed by naive Bayes in its calculation of an anomaly score. The actual anomaly score is the ratio of the logarithms of these two probabilities, as shown in Eqn 4.6.

$$A = \frac{\log P_3(S)}{\log P_3(N)} \quad (4.5)$$

$$= \frac{\sum_{i=0}^b (\log(s[c_i] + p) - \log(s_t + \alpha p))}{\sum_{i=0}^b (\log(n[c_i] + p) - \log(n_t + \alpha p))} \quad (4.6)$$

Without changing the value of A , we can simplify its expression by moving the constant logarithm calculation outside each of the sums, as shown in Eqn 4.7.

$$A = \frac{\sum_{i=0}^b \log(s[c_i] + p) - b \cdot \log(s_t + \alpha p)}{\sum_{i=0}^b \log(n[c_i] + p) - b \cdot \log(n_t + \alpha p)} \quad (4.7)$$

Now, let us suppose that k of these b commands are actually NBSCs. Since the anomaly score calculation is the same for any permutation of the same sequence of commands, we assume without loss of generality that these k commands are the last k commands in the block. I.e., commands c_{b-k}, \dots, c_b are NBSCs. Note that if c_i is a NBSC then, by definition, $s[c_i] = n[c_i] = 0$. The command has never appeared in the training data for either self or nonself. We can use this fact to isolate and simplify the role of NBSCs in the calculation of A as shown in Eqn 4.9.

$$A = \frac{\sum_{i=0}^{b-k-1} \log(s[c_i] + p) + \sum_{i=b-k}^b \log(s[c_i] + p) - b \cdot \log(s_t + \alpha p)}{\sum_{i=0}^{b-k-1} \log(n[c_i] + p) + \sum_{i=b-k}^b \log(n[c_i] + p) - b \cdot \log(n_t + \alpha p)} \quad (4.8)$$

$$= \frac{\sum_{i=0}^{b-k-1} \log(s[c_i] + p) + k \cdot \log(p) - b \cdot \log(s_t + \alpha p)}{\sum_{i=0}^{b-k-1} \log(n[c_i] + p) + k \cdot \log(p) - b \cdot \log(n_t + \alpha p)} \quad (4.9)$$

From this equation, it would appear that the effect of NBSCs is equal in both the numerator and the denominator – that is, an NBSC will have the same effect on both the self probability and the nonself probability. As such, one might assume that an NBSC will have a neutral effect, rather than the hypothesized effect of lowering the anomaly score.

However, such an assumption does not take into account the interaction between NBSCs and other aspects of the calculation of the anomaly score. Consider the case, as in all the evaluation datasets we have used, where the same number of commands is collected to be used as the training data for each user. E.g., naive Bayes is trained using 1000 commands from each user. In such cases, the total number of nonself training commands is going to be an integer multiple of the total number of self training commands. If we call this integer m , then $n_t = m \cdot s_t$. E.g., if there are 50 users and 1000 commands from each user then $s_t = 1000$ and $n_t = 49000 = 49 \cdot s_t$. In this example, $m = 49$. In the calculation of A , we can replace n_t with $m \cdot s_t$, as shown in Eqn 4.10.

$$A = \frac{\sum_{i=0}^{b-k-1} \log(s[c_i] + p) + k \cdot \log(p) - b \cdot \log(s_t + \alpha p)}{\sum_{i=0}^{b-k-1} \log(n[c_i] + p) + k \cdot \log(p) - b \cdot \log(m \cdot s_t + \alpha p)} \quad (4.10)$$

We have effectively separated the numerator and the denominator each into three parts: the effect of seen commands, of NBSCs, and of the size of the training data. The first part, the effect of seen commands, is the summation, and it calculates the effect of the commands that appear in the self and nonself data on the anomaly score. If most of the commands appear infrequently in the self training data and frequently in the nonself, the effect will be a large negative number in the numerator and a small negative number in the denominator. As intuitively expected, this will lead to a large anomaly score. The effect of this first part will vary depending on the type and character of the commands in the block.

The second part is the effect of NBSCs. Each NBSC adds an equal value to both the numerator and the denominator. The effect of this part will vary in magnitude depending on the number of NBSCs, but the effect will be the same on both the numerator and the denominator.

The third part is the effect of the amount training data. This part exerts a constant effect on each test block. Perhaps surprisingly, its effect is to decrease the nonself probability with respect to the self probability. Consider starting with the same amount of self and nonself training data (i.e., $m = 1$). In this case, the effect of this part of the calculation is the same on both the numerator and the denominator. However, as the number of nonself users increases and the amount of nonself training data grows, so too does this effect on the denominator. This part of the denominator becomes a large negative number which, in effect, increases the anomaly score.

Because of this third part, there is a natural bias toward a self probability. If all the commands in the block appear with the same frequency in both the self and the nonself data, the denominator will still be a more negative number than the numerator. As such, all else equal, the estimated self probability will be greater than the estimated nonself probability. Now, the important aspect to note with respect to the effect of NBSCs is that, in a block of commands full of NBSCs, its effects on the self and nonself data are equal, and so this natural bias toward higher self probability takes over.

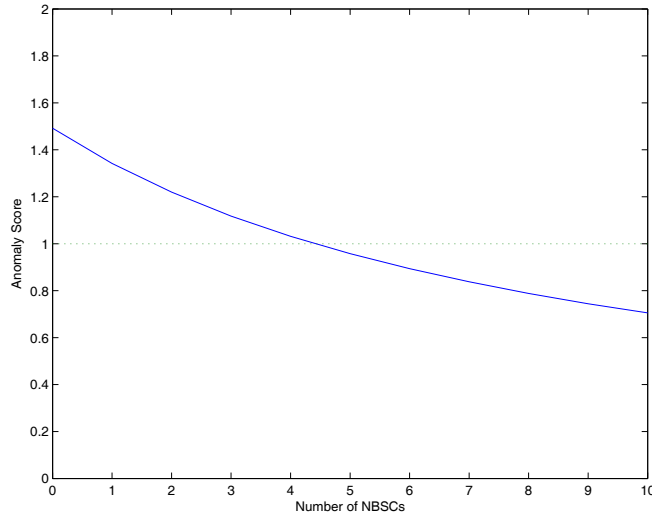


Figure 4.1: Graph of the relationship between the number of NBSCs and the anomaly score in the mathematical model of the naive Bayes algorithm.

To support the correctness of this analysis, we graph the anomaly score A as a function of the number of NBSCs (k) in Figure 4.1. Since the anomaly score depends on factors other than the number of NBSCs, we chose a block size of 10. We set the number of commands in the self training data (s_t) to 1000 and the number of nonself users (m) to 49. We set the pseudocount (p) to 0.1 and the alphabet size (α) to 300. We assume that in addition to the NBSCs, there was one other command in the test block and that it was repeated as many times as necessary to fill the block. We assumed the command appeared zero times in the self training data ($s[c_i]$) and 100 times in the nonself training data ($n[c_i]$). The number of NBSCs in the block range from 0 to 10.

Figure 4.1 shows that, without any NBSCs, the anomaly score for this block is higher than 1, indicating that the nonself probability is greater than the self probability. As the number of NBSCs increase, the anomaly score drops, indicating that the difference between the self and nonself probabilities is decreasing. At five NBSCs, the self probability becomes higher than the nonself probability, as indicated by the anomaly score dropping below one. The score continues to drop as the number of NBSCs increases to ten. This graphical demonstration supports the correctness of our analysis, and confirms that the effect of NBSCs is to lower the anomaly score by increasing the self probability with respect to the nonself probability.

4.3.2 Summary

In this section we have answered the question of why NBSCs have the hypothesized effect of decreasing the anomaly score calculated by naive Bayes. Due to the large amount of nonself training data as compared to the self training data, naive

Bayes is initially biased in favor of self probability when a command appears that has not been seen. If enough NBSCs appear in the command block, this bias becomes great enough to lower the anomaly score. This discovery was made through the construction of a mathematical model describing the calculations performed by naive Bayes as it makes its decision as to whether a block of commands looks legitimate or suspicious. This discovery supports our hypothesis that NBSCs tend to lower the anomaly score.

4.4 Empirical study using synthetic data

Is this mathematical model accurate? Does the theoretical result carry over to results with actual data? Do these results hold in a variety of environments with varying characteristics? In this section, we describe how synthetic data was generated with diverse features and how, in each dataset, NBSCs were introduced. We observe the performance of naive Bayes and how it changes as the number of NBSCs increases. We show that under a wide variety of conditions, the effect of NBSCs is a tendency to lower the anomaly score such that blocks of commands which would otherwise be detected as masqueraders are instead missed.

Clearly, many parameters affect the performance of naive Bayes. In order to isolate the interaction between NBSCs and naive Bayes, we will need to carefully control other potentially confounding factors. Various factors may affect the influence of NBSCs, the most influential being the following:

- the number of nonself users,
- the similarity of the masquerade block to self, and
- the similarity of the masquerade block to nonself.

Some of these factors were recognized during previous work with naive Bayes; others, like the number of nonself users, were identified during the construction of a mathematical model of naive Bayes in the first stage of this investigation.

In this experiment, we select a range of values for each of these factors and synthesize different data sets to represent different values in this range. With this technique, we control for the confounding influences that these factors may cause.

To control for such factors, first we had to measure them. The measure of the number of nonself users is obvious, but the others require some explanation. The similarity of the masquerade block to self was measured as a percentage. First, for each of the 10 commands in the masquerade block, the relative frequency of each command in the self training data was calculated. Then, the average of these 10 relative frequencies is taken as the average self-similarity of the masquerade block. For example, suppose the masquerade block consisted of 10 repetitions of a single command. Suppose that the self training data consisted of 1000 commands and this command appeared 5 times. As such, the relative frequency of each command in the self training data is 0.5% (or $5/1000$). Since this one command is repeated 10 times in the block, the average relative frequency for the block is also 0.5%. The similarity of the masquerade block to nonself was measured in similar terms, but

by calculating the frequencies using the nonself training data rather than the self training data.

The first step in data synthesis was to generate training data. We chose to consider between one and ten nonself users in the training data to see how variations in this parameter affected the experiment. The other two confounding factors – similarity to self and similarity to nonself – are measures of the masquerade block, not the training data. However, in order to afford a variety of values for this measure, we must synthesize training data that has commands that occur with a variety of frequencies. For example, we want commands that constitute 1% of the self training data but only 0.1% of the nonself training data. We seed the training data with these commands so that when it comes time to construct a masquerade block with certain features (in the next step), we will have commands with those features with which to build the block.

We decided to seed the self training data with commands that appear with frequencies at every tenth of a percent between 0.0% and 1.0%. Similarly, we decided that the nonself data would be seeded with commands that appear with the same range of frequencies. Further, we made sure that for every combination of frequencies in this set (e.g., 0.7% in the self data and 0.4% in the nonself), there would be a command in both datasets that fits the bill. When possible, we try to use the same training data for multiple experiments. The reason for this is that changes to the training data affect some parameters that naive Bayes calculates based on the training data, e.g., a threshold for the anomaly score is calculated using five-fold cross validation on the training data. Clearly, when the number of nonself users changes, so must the training data. However, once the number of nonself users was fixed, we were able to construct a set of self and nonself training data that accommodated all the self and nonself similarity possibilities under consideration. For each user’s training data, either the self user or the one or more nonself users, we generated 1000 commands of training data. We generated 10 training datasets, each with a different number of nonself users between 1 and 10 inclusive.

The second step, once the training data was synthesized, was to synthesize a masquerade block. The masquerade block consists of a block of 10 commands. From the training data, we have hundreds of commands available each with different features. Some commands constitute 1% of the self training data but never appear in the nonself training data. Other commands constitute 0.3% of the self training data and 0.6% of the nonself. As described above, we have constructed the training data such that it contains commands that occur with every combination of frequencies between 0.0% and 1.0% (at 0.1% intervals) in both the self and nonself training data. Since there are 11 possible values for a command’s self frequency and 11 possible values for its nonself frequency, there are 121 commands in the training data which occur with a precisely determined, known frequency.

We decided to construct masquerade blocks by choosing one of these 121 commands that occur with a particular frequency and by repeating it 10 times to fill the masquerade block. To effect this strategy, we synthesized 121 masquerade blocks for each of the 10 sets of training data, so there are 1210 distinct datasets at the end of this stage.

The third step, once we have synthesized training data and masquerade blocks

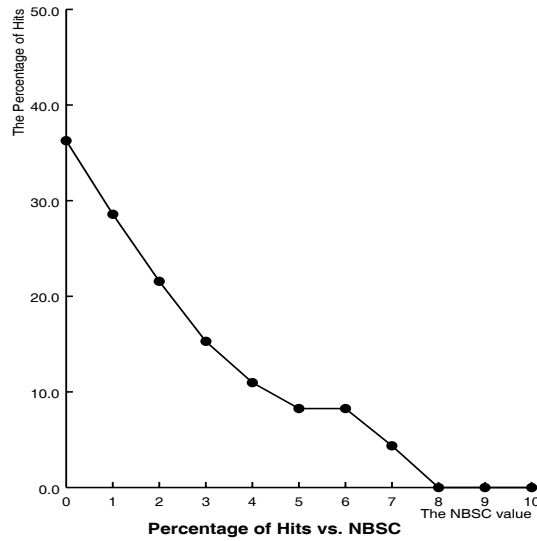


Figure 4.2: Graph of the effect the number of NBSCs has on the percentage of synthetic datasets in which the masquerader block was correctly detected.

is to inject NBSCs into the masquerade blocks. The masquerade blocks are 10 commands long and so we have the option either of injecting no NBSCs or of injecting between 1 and 10 NBSCs. Since the order of commands in the masquerade block do not matter, but the size of the block must remain fixed at 10, for each NBSC we inject, we must choose one command to remove from the block. We chose to try every combination of injections and so, from each of the 1210 datasets at the end of the last step, we generated 11 more datasets (one for each distinct injection of some number of NBSCs). At the end of this step, we have 13310 datasets.

The final step is to run naive Bayes on each of these datasets. On each, naive Bayes was first trained on the synthetic self and nonself training data. It was then fed the masquerade block (with the appropriate number of NBSC injections). At the end of each run, we record both the anomaly score calculated for the block and also whether naive Bayes correctly detected the block as a masquerader or not.

4.4.1 Findings

Across the runs, the NBSC count increases from 0 to 10. At each count, there are 1210 runs. Figure 4.2 shows the percentage of these runs in which the masquerade block is detected as the number of NBSCs increases. Initially, with no NBSCs, naive Bayes is able to detect the masquerader in approximately 36% of the runs. With only a single NBSC in the masquerade block, that percentage drops to about 29%. As the number of NBSCs increases, the hit rate falls. When the masquerade block consists of eight or more masqueraders, the percentage of blocks correctly detected drops to zero.

Our hypothesis concerns not just naive Bayes' ability to detect masquerade

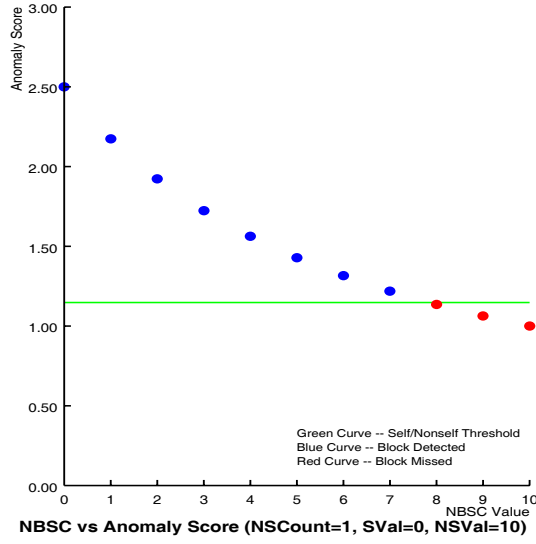


Figure 4.3: Graph of the effect of NBSCs on the anomaly score of a masquerade block when there is one nonself user, the commands in the masquerade block never appear in the self training data and appear with a relative frequency of 1.0% in the nonself training data.

blocks but also the effect of NBSCs on the anomaly score calculation that underlies this ability. Figure 4.3 plots the anomaly score of a particular masquerade block as the commands in the block are replaced by NBSCs. The particular block was taken from a dataset containing only 1 nonself user. The masquerade block itself contained 10 repetitions of a command which never appeared in the self training data and appeared with a relative frequency of 1% in the nonself training data. In the figure, the plotted points show a drop in the anomaly score as the number of NBSCs increases. The color of each point indicates whether it was correctly detected; blue means that the masquerader was detected, while red means the masquerader was missed. The green horizontal line indicates the threshold naive Bayes uses to decide whether an anomaly score is high enough trigger an alarm about a possible masquerader.

While this one example supports our hypothesis about the effect of NBSCs on naive Bayes, we consider whether this holds in general. In Figure 4.4, we plot the anomaly scores of all 121 masquerade blocks for which the number of nonself users was 10. For each masquerade block, we plot the anomaly score as commands in the block are replaced with NBSCs. The 121 blue lines are the 121 anomaly scores. As the figure shows, no matter the initial anomaly score, the injection of NBSCs causes the anomaly scores to converge to a point at about 0.83, well below the threshold at which naive Bayes considers the block to be suspicious. Once again, NBSCs tend to lower anomaly scores such that masquerade blocks become undetectable.

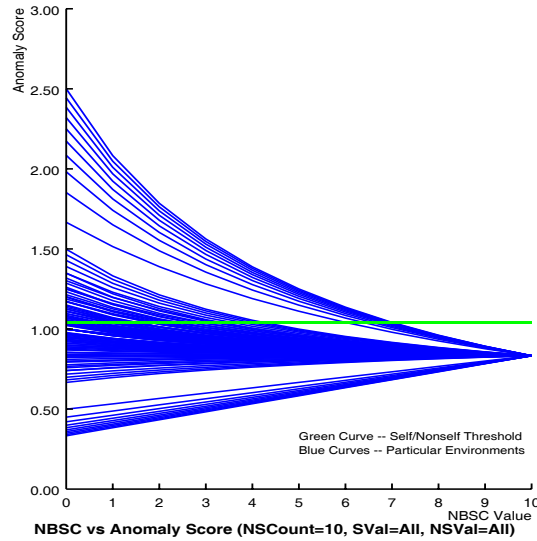


Figure 4.4: Graph of the effect of NBSCs on the anomaly score of all 121 masquerade blocks for which the number of nonself users was 10. The commands used in the masquerade blocks appear with in the self training data with frequencies ranging from 0.0% to 1.0% and appear in the nonself training data with the same range of frequencies.

4.4.2 Summary

In this section, we confirmed that NBSCs have the predicted effect on naive Bayes under a wide variety of conditions. We identified several confounding factors which also have an effect on naive Bayes ability to detect masqueraders. We synthesized data in which each of these factors is controlled. The results have shown that taking all of these confounding factors into account, increasing the number of NBSCs in the command block reduces the ability of naive Bayes to detect the block. These results add further support to the hypothesis that NBSCs tend to lower the anomaly score and thereby cause naive Bayes to miss masqueraders that would otherwise have been detected.

4.5 Mitigating the effect of NBSCs

How do we deal with this potential vulnerability in the interaction between NBSCs in the data and the naive Bayes algorithm? If a masquerader were aware of this effect of NBSCs, it would be in his or her interest to consider the commands used in an attack to make use of ones that quite possibly have never been used before. This may or may not be an easy task depending on the environment, but it is certainly possible in many computing environments to create new commands which, with great likelihood, have never before been seen.

In this section, we consider a strategy for mitigating the risk posed by this effect of NBSCs on naive Bayes. Our strategy is to create a NBSC detector that solely

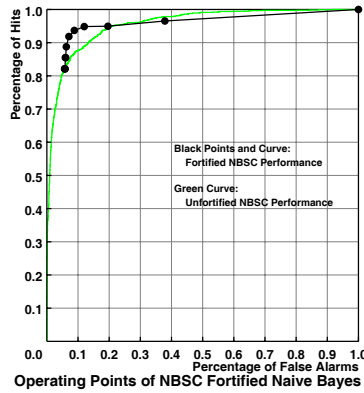


Figure 4.5: Graph of the ROC curves for both naive Bayes (green) and the new masquerade detector fortified with a NBSC detector (black).

detects NBSCs. This detector can be integrated into the masquerade detector so that masqueraders who intersperse their actions with NBSCs to avoid detection by naive Bayes will be detected by this supplemental NBSC detector instead. Once we create this detector, we evaluate it using the dataset from the enrichment experiment that initiated this investigation. We compare the results of this new detector against those of the regular naive-Bayes-based detector.

As designed, the NBSC detector trains on the same data as Naive Bayes (i.e., the self and nonself training data). However, the NBSC detector makes no distinction between self and nonself data. It simply keeps track of which commands occur in either dataset and which commands do not. Once the detector has trained, it is fed the same masquerade blocks as naive Bayes. However, the NBSC detector simply counts the number of NBSCs in the block and calculates their relative frequency within the block. If that relative frequency is above a threshold percentage (e.g., 50%), the detector issues an alarm.

4.5.1 Findings

The first thing to do with the detector was to identify a threshold frequency at which to generate an alarm. To do so, we swept through a range of values from 0% to 100% and plotted the performance of the detector in terms of its hit and false alarm rate. We plotted these points on a receiver operating characteristic (or ROC) curve. We also plotted the ROC curve for naive Bayes without this NBSC detector to measure the change in performance. These two ROC curves are shown in Figure 4.5.

As the threshold decreases from 100%, we immediately see gains in the hit rate of the new masquerade detector.

With a threshold of 50%, we obtain a hit rate of 88.9%, well above the 82.1% hit rate at which naive Bayes functions. With this threshold, both of the so-called *super masqueraders* – who were undetectable no matter which user they impersonated – are correctly detected. The new masquerade detector correctly detects 100

masquerade blocks missed by the original naive Bayes. About 38% of the masqueraders missed by naive Bayes were detected by the new masquerade detector, yet the new detector was responsible for only 27 more false alarms, an increase from 5.7% to 6.3%.

4.5.2 Summary

In this section, we described a technique for mitigating the adverse effects of NBSCs on the performance of naive Bayes. This technique was designed to specifically address the weakness uncovered during the earlier stages of this investigation. In our evaluation, this new detector was responsible for only a small increase in the hit rate, from 5.7% to 6.3% while 38% of the masqueraders missed by naive Bayes were able to be detected with the new detector.

4.6 Conclusion

In our investigation, we have identified a previously unknown interaction between the characteristics of data and the architecture of a masquerade detection system. This interaction made it possible for a masquerader to avoid detection by interspersing his or her activities with so called never-before-seen commands (NBSCs) to thwart the naive Bayes algorithm which has proven useful in masquerade detection. During this investigation, we have provided compelling evidence to support the hypothesis that NBSCs tend to lower the naive Bayes anomaly score such that it drops below the threshold score, causing decisions to favor the user (self) instead of the masquerader (nonself). This behavior results in missed alarms. First, we examined the mathematical underpinnings of the naive Bayes algorithm in order to show a theoretical justification for the observed effect of NBSCs. Second, we empirically measured the effects of NBSCs while carefully controlling for confounding factors in the environment. Third, after compiling strong evidence to support the hypothesis, we examined what could be done to mitigate the adverse effect of these never-before-seen commands. We showed that interactions between characteristics of data and the detector architecture do exist. We also showed how adverse effects of these interactions on the performance of the detector can be mitigated once they are identified.

Task 5

Review of facts acquired and lessons learned in developing a science base for anomaly detection

This task was de-scoped and omitted due to sponsor exigencies.

Task 6

Program code and documentation for the CSD anomaly detector

Proem. Program code and documentation is presented for one implementation from one of the following classes of anomaly-detection algorithms: neural network detectors, probabilistic detectors, sequence time-delay embedding detectors, and frequentist detectors.

6.1 Introduction

The sequence, time-delay, embedding anomaly detector described in this chapter analyzes the sequential relationships between elements of categorical data in order to effect anomaly detection, the detector is referred to as the “Congruent Sequence Detector” or CSD for short. The CSD determines the abnormality of a sequence by how much it differs from the sequences found in the data that characterizes the normal behavior of the monitored subject (more detailed information about the detector, its algorithm and performance characteristics can be found in chapter 8). A monitored subject may be a human user, a system program or any process that can be audited for data that describes the operation of that process, and the categorical data analyzed by the detector may come from any sensor, e.g., BSM, strace, tcpdump, process accounting and so forth. The source code for the CSD anomaly detector is in the Task 6 directory of the accompanying CD/DVD.

6.2 Description

The CSD takes as input a training file, a test file, and a key file, and produces as output the results of the anomaly detection process.

The main parameter for the CSD is the sequence size. Given a sequence size X , the CSD will partition the training data into a “normal” library of sequences of size X . The size X sequences that populate this “normal” library, are obtained by sliding a window of size X across the training data where each unique size- X sequence encountered is stored in a database.

After training is completed, the detector begins processing the test data. Using a sliding window of the same size, X , each sequence from the test data is compared to every sequence in the “normal” library of sequences previously established using the training data. A “similarity measure” determines the degree to which a given test sequence deviates from the normal sequences. The response of the anomaly detector can be summarized as such: given two fixed-length sequences of the same size, the similarity metric assigns a value (similarity score) between 0 and $DW(DW + 1)/2$, where DW is the size of the sequence and 0 denotes the greatest degree of dissimilarity (anomalous) and $DW(DW + 1)/2$ denotes the greatest degree of similarity. Therefore, given two fixed-length sequences of size 3 for example, a complete mismatch between the two sequences will register a similarity score of 0. If the two sequences were identical (normal) then a similarity score of 6 will be registered. For more details of the CSD, refer to the report for Task 8.

There are three inputs to the CSD. The first is a file containing training data in the following format:

```
6
1
2
3
4
5
6
1
2
3
4
5
6
1
2
3
```

The data should be entered as one element of categorical data per line. Numbers are used to represent categorical elements and as such, the intrinsic value of the numbers themselves have no significance.

The second input to the CSD is a file containing test data in the same format - one categorical element per line, and the third is the key file. The following is an example of the contents of a key file.

```
677 680
2855 2858
4225 4228
5193 5196
6932 6935
```

```

      8764  8767
10281 10284
12256 12259
13754 13757
14557 14560
16926 16929
17978 17981
20136 20139
22018 22021
22840 22843

```

The first number denotes the line number in the test file where the anomaly begins, and the second number is the line number in the test file where the anomaly ends. Therefore in the first line of the example above, an anomaly of size 4 was injected into the test file on line 677. The anomaly ends on line 680. The second anomaly begins at 2855, and ends at 2858, and so forth.

6.3 Usage

The executable for the CSD is labeled “csd,” and currently works on Linux. To use the CSD on other platforms, simply recompile by running:

```
make
```

A “Makefile” is included in the package.

The detector is invoked in the following manner:

```

csd -f <training file>
    -t <test file>
    -k <key file>
    -w <window size>
    -h <threshold>
    -q

```

where

- training file is the name of the file containing the training data,
- test file is the name of the file containing the test data,
- key file is the name of the file containing the location of the anomalies in the test data,
- window size is the desired sequence size, and
- threshold is the value that defines a sequence to be anomalous. Recall that 0 denotes the greatest degree of dissimilarity (anomalous) and $DW(DW +$

1)/2 denotes the greatest degree of similarity. The threshold is a value chosen from between these two limits. So if the threshold is set at 4, sequences with similarity scores less than 4 will register as anomalous (since the closer the score gets to 0 the more anomalous the sequence), sequences with scores at or greater than 4 will register as normal.

If the -q flag is set then the program will provide a summary of the anomaly detection process instead of the normal output. For example, executing the following command with the -q flag:

```
./csd -f train -t test -k key -w 3 -h 6 -q
```

will produce the following to standard output:

```
train_filename: train
test_filename:  test
key_filename:   key
window_size:    3
threshold:      6
Initializing...
...Initializing complete
Learning training data...done.
Tupletizing test data...done.
Hit ratio       : 100.00\%
Miss ratio      :  0.00\%
False alarm ratio:  0.00\%
Total anomalies :  305
```

The output lists the hit, miss and false alarm rates obtained from the given input files.

The CSD outputs to standard output. Therefore:

```
./csd -f train -t test -k key -w 3 -h 3
```

will result in “csd” taking the training file called “train”, the test file called “test” and the key file called “key”, to produce the following results to standard output (note that the files “train”, “test” and “key” are also included in the package):

```
train_filename: train
test_filename:  test
key_filename:   key
window_size:    3
threshold:      3
Initializing...
...Initializing complete
Learning training data...done.
Tupletizing test data...done.
      0:      4      [  6]
```

```

1:      1      [ 6]
2:      2      [ 6]
3:      3      [ 6]
4:      4      [ 6]
5:      5      [ 6]
6:      6      [ 6]
7:      1      [ 6]
8:      2      [ 6]
9:      3      [ 6]
10:     4      [ 6]
11:     5      [ 6]
12:     6      [ 6]

```

The example above shows the first 22 lines of the detector's output. The first 9 lines serve diagnostic purposes, namely to verify where the data came from, i.e., from the files train, test and key, what parameters were used for this instance of execution, i.e., sequence size of 3 and threshold of 3, and the success or failure of the detector's internal processes. The last 13 lines are the output of the detector.

The first of the 13 lines

```

0:      4      [ 6]

```

is deciphered as such: the first number denotes the line number of test file, the numbering system begins at 0 therefore the first line of the test file is line 0, the second line of the test file is line 1, and so forth.

The second number is the raw input, i.e., the actual categorical element that can be found on the first line of the test file, which in this case is 4.

The final number is the similarity score assigned to the sequence of size X beginning on the first line of the test file and ending on the Xth line of the test file, where X is a user defined parameter.

Anomalies are marked by asterisks. In this example, the key file says that an anomaly was injected on line 677 ending on line 680. The output of CSD for line 677 is as follows:

```

672:     3      [ 6]
673:     4      [ 6]
674:     5  -    [ 3]
675:     6 * -    [ 1]
676:    11 * -    [ 0]
677:    24 * -    [ 0]
678:    24 * -    [ 1]
679:    19 * +    [ 3]
680:     5 * -    [ 6]
681:     6  -    [ 6]
682:     1      [ 6]
683:     6      [ 6]

```

The elements 11, 24, 24, 19 comprise the anomalous sequence found in the test file. The asterisk marks those sequences with similarity scores less than the threshold 3. So for example, the size three sequence 4,5,6 on lines 673,674 and 675 computed a similarity score of 1 which is less than three and hence an asterisk is placed to mark that anomalous sequence.

The symbol “-” simply marks those sequences that produced a similarity score less than the most normal score, which in this case is 6. The sequences marked with “-” need not necessarily be the most anomalous sequences, i.e. those with scores of 0.

The symbol “+” simply marks the point at which the similarity score jumps from less than the most normal back to normal again. This is to help identify the “edges” of anomalous sequences for subsequent analysis.

Task 7

Test data suites used in ascertaining the efficacy of the CSD algorithm

Proem. Test data are presented for use in ascertaining the efficacy of the CSD detection algorithm.

7.1 Introduction

This chapter describes a series of benchmark data-sets that were used to evaluate the CSD, an anomaly detector that functions by determining the similarity of fixed-length sequences of categorical data. (This anomaly detector is described in detail in elsewhere in this report, as is the detector's operational efficacy, as based on the data-set described here.) These data sets are available in the Task 7 directory of the accompanying CD/DVD.

The data-sets were designed to evaluate anomaly detectors based on fixed-length sequences. The focus of the data-sets are the injected anomalies. The aim is to determine the ability of an anomaly detector at detecting anomalies with well-defined characteristics. Since these anomalies may be the manifestations of attacks, faults or other events of interest, it is important to evaluate a detector's sensitivity toward them. The anomalies injected into the data-set are rare sequence anomalies composed of rare sub-sequences. A rare sequence is defined as a sequence with a relative frequency of less than 0.005 in the training data.

7.2 Description

There are 7 benchmark data-sets in 7 directories. The directories are labeled, SS2, SS3, SS4, SS5, SS6, SS7, and SS8, "SS" stands for "Sequence Size". SS2 is therefore the data-set used to evaluate sequence-based anomaly detectors employing a sequence size of 2.

Within each of the 7 directories are three files

- train - this directory contains training-data files.
- test - this directory contains test-data files.
- key - this directory contains files that list the exact location of the injected anomalies (ground-truth).

The following is an example of the contents of a file containing training data:

```
3
4
5
6
7
8
1
2
3
4
5
6
7
8
```

There is one element of categorical data per line. Numbers are used to represent categorical elements and, as such, the intrinsic value of the numbers themselves have no significance.

The training and the background data (i.e. the test data without injected anomalies) was carefully designed to eliminate noise. This was done by constraining the regularity of the data to 0.1. The most regular data measures at 0.0, therefore 0.1 is the second most regular value. The unpredictability of data with regularity measures higher than 0.1 may result in spurious or unintended anomalies outside the known anomalies injected into the background data. Extremely regular data with an anomaly measure of 0.0 was not used in creating this data-set due to the characteristics of anomaly injected into the test files. The anomalies injected into the test files are rare sequence anomalies. These are among the more difficult anomalies to detect, detectors such as stide, for example, would not be able to identify rare sequence anomalies at all. Anomalies are created using training data as the base. For example, sequences foreign to the training data are used as foreign sequence anomalies, and sequences that are rare in the training data are used as rare sequence anomalies. In the case of training data measuring 0.0 on the regularity scale, rare sequences will be unavailable for constructing anomalies because every sequence occurs as frequently as every other. Under these circumstances there can be no rare sequences for generating anomalies.

The following is an example of the contents of a file containing test data, the format is identical to that of the training data:

```
5
```

6
7
8
1
2
3
4
5
6
7
8
1
2
3
4
5

The following is an example of the contents of a key file.

135019 135026 4 1 2 3 4 7 6 7

The first number is the line number where the anomaly begins, the second number is the line number where the anomaly ends and the remainder is the actual anomaly itself that was injected into the background data. Therefore in the first line of the example above, an anomaly of size 8 was injected into the test file at line 135019 ending on line 135026.

The anomalies injected into the data-set are rare sequences that are 8 elements long. The anomalies have relative frequencies of less than 0.005 in the training data, and the sub-sequences of each size-8 anomalies are themselves rare sequences. Note however, that the size-8 anomalies in directory “SS2” will have rare size-2 sub-sequences, the size-8 anomalies in directory “SS3” will have rare size-3 sub-sequences, and so forth. The size of the sub-sequences are identical to the sequence size used by the anomaly detector because a detector window of a fixed size, X , sliding over an anomaly will only “see” sub-sequences of size X .

Task 8

Strengths and weaknesses of the CSD detection algorithm

Proem. A selection of strengths and weaknesses of the CSD detection algorithm is presented, including detailed performance results.

8.1 Introduction

The “Congruent Sequence Detector” (CSD) described in this document functions by analyzing the sequential relationships between elements of categorical data in order to effect anomaly detection. The CSD determines the abnormality of a given sequence by how much it differs from the sequences found in the data that characterizes the normal behavior of a monitored subject. A monitored subject may be a human user, a system program or any process that can be audited for data that describes the operation of that process, and the categorical data analyzed by the detector may come from any sensor, e.g., BSM, strace, tcpdump, process accounting and so forth.

This chapter will first provide a description of the algorithm, followed by a description of the evaluation methodology, then a presentation of the results, a discussion, and finally the conclusion in terms of the lessons learned.

8.2 Description

The CSD acquires its model of normal behavior by partitioning training data into fixed-length sequences. This is done by sliding a detector window of size DW over the training data. Each size- DW sequence obtained from the data stream is stored in a dictionary of sequences of length DW .

A similarity metric is then used to establish the degree of similarity between each DW -sized sequence in the test data to the normal DW -sized sequences acquired by the anomaly detector from the training data. The fixed-length, overlapping sequences from the test data is obtained by sliding a size DW detector window over the entire test data stream. The size of the window used to segment the test data is the same as size of the window used to segment the training data.

For every fixed-length sequence of size DW , obtained from the test data, a metric is employed to determine the degree of similarity between the fixed-length sequence from the test data, and the model of normal behavior captured by the dictionary. This similarity metric involves a function that takes two fixed-length sequences and returns a similarity value between 0 and $DW(DW + 1)/2$. The more similar the two fixed-length sequences are, the larger the similarity value. The less similar, the smaller the similarity value. A similarity value of 0 denotes the greatest degree of dissimilarity between two fixed-length sequences. The similarity metric is biased in support of adjacent matches with an upper bound that is polynomial in the length of the sequences.

A more formal description can be described as such.

Let DW be the length of a sequence. The model of normal for the CSD detector is obtained by sliding a window of size DW over the training data; each unique size- N sequence encountered is stored in a database referred to as the dictionary.

For every fixed-length sequence of size DW obtained from the test data, the following similarity metric is used to provide a numerical indication of how similar the fixed-length sequence is to each of the size- DW sequences stored in the dictionary of normal behavior.

For length DW , the similarity between two given sequences,

$$X = (x_0, x_1, \dots, x_{DW-1}) \text{ and}$$

$$Y = (y_0, y_1, \dots, y_{DW-1})$$

is defined by the pair of functions:

$$w(X, Y, i) = \begin{cases} 0 & \text{if } i < 0, \text{ or } x_i \neq y_i \\ 1 + w(X, Y, i - 1) & \text{if } x_i = y_i \end{cases}$$

where $w(X, Y, i) = 0$ for $i < 0$, so that $w(X, Y, 0)$ is well defined when $x_0 = y_0$, and

$$Sim(X, Y) = \sum_{i=0}^{DW-1} w(X, Y, i)$$

This metric yields a higher score for more similar sequences, bounded between 0 and $DW(DW + 1)/2$, where DW is the sequence length. As shown in the equations above, the metric is biased towards adjacent elements.

Note that there is more than one sequence of size DW in the normal dictionary. Therefore, in order to determine the similarity between a single sequence from the test data, the test sequence, and the entire normal dictionary, the single test sequence of size DW is compared to each and every one of the sequences of size DW in the normal dictionary using the similarity metric laid out above. The similarity of a single sequence from the test data, Seq_i , to a set of sequences L in the normal dictionary is defined to be:

$$Sim(Seq_i, L) = \max_{Seq_j \in L} \{Sim(Seq_i, Seq_j)\}$$

The similarity value assigned to a sequence is a measure of how similar that sequence is to the *most* similar sequence in the normal dictionary.

The anomaly signal for the CSD is obtained by applying a windowed mean-value filter, a smoothing filter, to the raw similarity values, which at sequence i of the input stream is defined by:

$$m_w(i, L) = \frac{1}{w} \sum_{j=i-w}^i Sim(Seq_j, L)$$

where L is the normal dictionary, and w is the window length. Like the locality frame in stide, the window length w is a completely separate parameter to the detector-window size parameter. Each time a size- DW sequence is obtained from the test data, the similarity of that sequence to the all the sequences in the normal dictionary is calculated. Then the average of the similarity values of the past X sequences is determined, where X is the size of the smoothing window w . This average serves as the anomaly signal.

8.3 Evaluation Methodology

Two experiments were designed to determine the performance strengths and weaknesses of the CSD. In the first experiment, the size of the detector window was set to equal the size of the anomaly. The aim was to identify which of the three anomaly classes (the foreign-symbol anomaly (FF_{AS}), the foreign-sequence anomaly (FA_{AS}) or the rare-sequence anomaly (RA_{AS}), AS refers to the anomaly size, hence RA_{AS} refers to a rare-sequence anomaly of size AS) were detectable under such conditions. The three anomalies listed were selected because they are visible to sequence-based anomaly detectors. Each of these anomalies can be defined as such:

- The foreign-symbol anomaly

An anomalous sequence where no element within the sequence belongs to the training-set alphabet. The training-set alphabet is the set of unique symbols in the training data. For example, given a training-set alphabet comprised of A B C D, the sequence W X Y Z would be a foreign-symbol anomaly of length 4.

- The foreign-sequence anomaly

An anomalous sequence of length N where each individual element within the sequence is a member of the training-set alphabet, but where the entire length- N sequence itself does not occur in the training data. For example, given a training-set alphabet comprised of A B C D, the sequence A A A A may be a foreign-sequence anomaly because although the symbol A belongs to the training-set alphabet, the sequence A A A A does not occur in the training data.

Note: Since foreign-sequence anomalies are concerned with the order of elements within a sequence, it does not make sense to have a foreign-sequence anomaly of length 1. A single element that belongs to the training-set alphabet is not a sequence of more than one element whose positions relative

to each other can be described as foreign. A single element that does not belong to the training-set alphabet is a foreign-symbol anomaly of size 1.

- The rare-sequence anomaly

Rare-sequence anomalies are length- N sequences that infrequently occur in the training data. Each individual element in the length- N sequence is a member of the training-set alphabet, and the sequence of N elements itself exists in the training data. The infrequency with which a sequence occurs in the training data marks the rarity of that sequence. The frequency of occurrence is typically measured as relative frequency, and the user specifies a *rare threshold* which is a relative frequency that identifies rare sequences. A trivial example is one where the rare threshold is set to be a relative frequency of 1%, causing sequences with relative frequencies of 1% or less to be considered rare.

It can be argued as to whether rare sequences are really anomalous. Rare sequences, unlike foreign sequences, may also be present in data representing normal behavior, and as a consequence can be considered less likely to be anomalous. However, since there are anomaly detection algorithms in the intrusion detection literature that have been designed around the idea that rare sequences are anomalous and are therefore useful indicators of faults [42], this study will incorporate rare sequences as a type of anomalous phenomenon against which anomaly detection effectiveness should be evaluated.

The second experiment varied the size of the detector window with respect to the size of the anomaly, and observed how many instances of the same anomaly class were detectable under various combinations of detector-window and anomaly size.

The first experiment was designed to establish detection coverage in terms of how many classes of anomalies were detectable and the second experiment was designed to establish detection coverage in terms of how many instances of the same class of anomaly were detectable when the detector-window size was varied with respect to the anomaly size. If it is true that all anomaly detectors are equally capable at detecting anomalies, then the results (detection coverage) for both experiments, for all the anomaly detectors, are expected to be identical. However, if it is not true that all anomaly detectors are equally capable of detecting anomalies, then differences are expected. Recall that the anomalies were defined specifically for detectors operating on fixed-length sequences of categorical data, and as such should be visible to the set of anomaly detectors being examined in this study. The only exception is the rare-sequence anomaly, this anomaly is expected to be visible only to those anomaly detectors that employ probabilistic methods in their detection schemes.

8.3.1 Experiment 1 - Standard Conditions

Aim

The aim of Experiment 1 was to identify which of the three anomaly classes, FF_{AS} , FA_{AS} and/or RA_{AS} , are detectable by each of the four anomaly detectors under standard conditions. Standard conditions refer to the size of the anomaly being equal to the size of the detector-window parameter.

Experiment 1 parameters

The following is an overview of the parameters used in Experiment 1.

- the sample size - 1,000,000.
- the alphabet size - 8.
- the size of the base anomaly (AS) - 8.
- the size of the detector window (DW) - 8.
- the list of anomalies injected into synthetic data:
 - FF_{AS} , a foreign-symbol anomaly.
 - $FA_{AS} \text{ CommonSeq}_{IDW} \text{ CommonSeq}_{BDW}$, a foreign-sequence anomaly composed of common internal sequences and common boundary sequences.
Internal sequences are those sequences that lie within the anomaly, and boundary sequences are those sequences that lie on the edge of the anomaly, i.e., sequences that contain some elements of the anomaly and some elements of the background data.
 - $RA_8 \text{ CommonSeq}_{IDW} \text{ CommonSeq}_{BDW}$, a rare-sequence anomaly composed of common internal sequences and common boundary sequences.
- a single anomalous incident injected into a single test data stream.

8.3.2 The data sets

There are four kinds of data that need to be generated for this experiment: the training data (data used to establish the normal context), the background test data (data without the anomalous event), test data (data with the embedded anomalous event), and the anomalous events themselves. These data sets were created for the sole purpose of identifying the detection capabilities of the anomaly detectors being studied in this thesis, the data were not generated to assist in the design or the creation of an anomaly detector that is capable of detecting the injected anomalies.

Constructing the training data

	1	2	3	4	5	6	7	8
1	0.004686	0.967200	0.004686	0.004686	0.004686	0.004686	0.004686	0.004686
2	0.004686	0.004686	0.967200	0.004686	0.004686	0.004686	0.004686	0.004686
3	0.004686	0.004686	0.004686	0.967200	0.004686	0.004686	0.004686	0.004686
4	0.004686	0.004686	0.004686	0.004686	0.967200	0.004686	0.004686	0.004686
5	0.004686	0.004686	0.004686	0.004686	0.004686	0.967200	0.004686	0.004686
6	0.004686	0.004686	0.004686	0.004686	0.004686	0.004686	0.967200	0.004686
7	0.004686	0.004686	0.004686	0.004686	0.004686	0.004686	0.004686	0.967200
8	0.967200	0.004686	0.004686	0.004686	0.004686	0.004686	0.004686	0.004686

Table 8.1: The training-data generator. The data generator is a transition matrix where the first column contains the current events and the first row contains the next event. Hence, if the current event is 3 then the probability that the next event is 4 is 0.967200.

The training data were generated using a transition matrix. A transition matrix is simply a table that contains the categorical values that may occur in the training data and the probabilities that govern the generation of each value given the previous categorical value. An example of a transition matrix and a more detailed explanation will be given below. Numbers were used to represent each element of the categorical data, i.e., the numerical value was ignored, and the numbers were treated as categories. The generation process was straightforward. The transition matrix was entered at random, where the random point of entry into the matrix was determined by a random number generator. Once inside the transition matrix, each transition was determined by a random number between 0 and 1. In order to permit the use of any random number generator, particularly one that has been certified to be as random as technically possible [24], 1,000,000 random numbers were generated first (1,000,000 is equal to the sample size of the training data). These random numbers were then stored in a table, and subsequently used to determine the matrix transitions. In this way, the random-number sequence can be retained, if need be, to enable a perfect repetition of the experimental sequence.

Table 8.1 displays the transition matrix that was used to generate the training data. The first column contains the current events, and the first row contains the next event. At the intersection of each row and column is the probability that the next event will follow the current event. For example, if the current event is 2 (the second item in the first column), then the probability that 4 will follow 2 is 0.004686. Similarly, the probability that 2 follows 1 is 0.967200.

This matrix was used to generate a stream of training data with a conditional entropy value of 0.1, and a sample size of 1,000,000 categorical elements. A conditional entropy value of 0.1 means that the training data was not perfectly predictable (regular). A data stream registering a conditional entropy value of 0 is the

most predictable and 1, the most irregular (random). The transition matrix chosen was the second most predictable data generator in the suite of 11 data generation transition matrices that ranged from the completely predictable to the random presented in [26] (also see [26, 18] for a more thorough treatment of predictability and the entropy measure). This transition matrix was chosen because it generated data with the following characteristics:

- a large proportion of the data consists of a repetition of the sequence 1, 2, 3, 4, 5, 6, 7, 8. E.g. 98% of a 1,000,000 element data stream generated with this transition matrix will consist of a repetition of the sequence 1, 2, 3, 4, 5, 6, 7, 8. This results in a consistent set of obviously common sequences regardless of sequence length. E.g., the sequence 2, 3 will be common, just as the sequence 3, 4, 5, and the sequence 6, 7, 8, 1, 2, and so forth. A pool of common sequences of various lengths that can be extracted from the training data was required to construct test data such that commonly occurring sequences made up the background data into which anomalies were injected. Such test data makes it possible to attribute the response of a detector to the injected anomaly rather than to any naturally occurring rare or foreign sequences.
- despite the repetition in a large portion of the data, resulting in a usable set of common sequences, there is a small amount of unpredictability in the transition matrix to ensure the occurrence of rare sequences that were necessary for the construction of rare-sequence anomalies.

The alphabet size for the training data was 8. Note that alphabet sizes in real-world data are certainly much higher than this, for example, the number of unique kernel calls in BSM audit data is approximately 243. The evaluation method was designed to assess the detection capabilities of an anomaly detector with respect to anomalies whose definitions will remain independent of changing data specifics such as alphabet size or sample size. For example, a foreign-sequence anomaly will remain a foreign-sequence anomaly with its associated characteristics regardless of changing alphabet size or sample size. Although alphabet size (or sample size) may play a role in, for example, influencing the number of unique or foreign sequences that occur in the data, the characteristics of a foreign sequence however, will remain constant and independent of changing alphabet size and sample size. Furthermore, defining the anomaly in a way that is independent of data characteristics, such as alphabet size, allows anomaly detection performance to remain consistent regardless of data set, synthetic or real-world. This point will be illustrated in Chapter 11, where anomaly detector performance and lessons learnt using synthetic data are validated using real-world data.

Constructing the test data

The test data was constructed in two stages:

- the background data for the test stream was first created, then
- the anomalies were constructed and injected.

Creating background data

The test data used in Experiment 1 was a combination of background data and injected anomalies. The background data consisted of the most commonly occurring sequences only, i.e., a repetition of the numbers 1, 2, 3, 4, 5, 6, 7, 8. The background data was composed in this way so that when any detector-window size, ranging from 1 to the sample size, is used, only common sequences are experienced by the detector. The background data was therefore clear of any spurious, naturally occurring foreign or rare sequences. However, as a consequence of this, the anomaly injection process had to be accomplished very carefully in order to keep intact the anomaly class, context sequences and background data as specified by the experiment.

Constructing and injecting anomalies

Experiment 1 required an example from each of the three anomaly classes, FF_{AS} , FA_{AS} and RA_{AS} . The following is a list of the anomalous incidents chosen as instances of each class of anomaly, including the reasons as to why they were chosen.

- FF_8

A foreign-symbol anomaly of length 8, $AS = 8$, composed of foreign internal sequences, where every internal sequence of sizes 2 to 7, $DW = 2..7$, is a foreign sequence, and foreign boundary sequences, where every boundary context sequence of sizes 2 to 15, $DW = 2..7$, is a foreign sequence. Recall that internal sequences are those sequences that lie within the anomaly, and boundary sequences are those sequences that lie on the edge of the anomaly, i.e., sequences that contain some elements of the anomaly and some elements of the background data.

- $FA_8 \text{ CommonSeqI}_{DW} \text{ CommonSeqB}_{DW}$.

A foreign-sequence anomaly of length 8, composed of common internal sequences, where every internal sequence of sizes 2 to 7, $DW = 2..7$, is a common sequence, and common boundary sequences, where every boundary sequence of sizes 2 to 15, $DW = 2..15$, is a common sequence. This anomalous incident was chosen because it is the cleanest example of a foreign-sequence anomaly in that only the size-8 sequence is foreign to the training data, all internal and boundary sequences are common sequences that already exist in the training data.

- $RA_8 \text{ CommonSeqI}_{DW} \text{ CommonSeqB}_{DW}$.

A rare-sequence anomaly of length 8, composed of common internal context sequences, where every internal sequence of sizes 2 to 7, $DW = 2..7$, is a

common sequence, and common boundary sequences, where every boundary sequence of sizes 2 to 15, $DW = 2..15$, is a common sequence. This anomalous incident was chosen because it is the cleanest example of a rare-sequence anomaly in that only the size-8 sequence is rare, all internal and boundary sequences are common sequences.

After the construction of the single stream of training data containing 1,000,000 categorical elements, the training data was used to create the specific anomalies described above. These anomalies were injected into the background data carefully so as not to cause perturbations in the background data surrounding the anomalies, i.e., no uncontrolled or unintended sequence types, e.g., foreign or rare, to occur as a result of the injection process.

Boundary sequences were most vulnerable in the injection process. Spurious rare or foreign sequences were most likely to occur in the boundary sequences where some elements of the injected anomaly and some elements of the background data combine within the sliding detector window to form, for example, foreign or rare sequences. To illustrate, Figure 8.1 shows the desired outcome of injecting an anomaly ($FA_{AS}ForeignSeqI_{DW}RareSeqB_{DW}$) into background data composed of DW sized common sequences. In contrast, Figure 8.2 shows the possible results of a random injection that resulted in a mixture of foreign and rare boundary sequences.

Experimental procedure for Experiment 1

The following is the outline of the method for Experiment 1:

- construct the training and test data.
- use the training data to create anomalies that are representative of each of the three anomaly classes.
- generate the background data for the test data.
- inject the anomalous incidents created in the previous step into test data streams.
- deploy each of the four anomaly detectors on the training and test data, setting the detector-window size to be the same as the size of the base anomaly, i.e., $DW = AS$.

8.3.3 Experiment 2 - Varying Conditions

Aim

The aim of Experiment 2 was to identify how many instances of a single class of anomaly that an anomaly detector successfully detected in Experiment 1, remained detectable under varying conditions. Varying conditions refer to the size of the detector-window being varied with respect to the size of the anomaly.



Figure 8.2: The undesired effect of introducing a foreign-sequence anomaly with foreign internal sequences and rare boundary sequences into background data consisting of common sequences only.

Experiment 2 parameters

The following is an overview of the parameters used in Experiment 2.

- the sample size - 1,000,000
- the alphabet size - 8
- the size of the base anomaly (AS) - 2 to 9
- the size of the detector window (DW) - 2 to 15
- the list of anomalies injected into synthetic data:
 - FF_{AS} , a foreign-symbol anomaly.
 - $FA_{AS} \text{ CommonSeq}_{IDW} \text{ CommonSeq}_{BDW}$, a foreign-sequence anomaly composed of common internal sequences and common boundary sequences.
 - $RA_8 \text{ CommonSeq}_{IDW} \text{ CommonSeq}_{BDW}$, a rare-sequence anomaly composed of common internal sequences and common boundary sequences.
- a single anomalous incident injected into a single test data stream.

The data sets

The construction of training and test data for Experiment 2 remained the same as for Experiment 1. The only difference was in the introduction of anomalous incidents with base anomaly sizes that ranged from $AS=2$ to 9. Experiment 1 only used base anomalies of size 8.

Experimental procedure

The following is the outline of the method for Experiment 2:

- using the training data set from Experiment 1, for each of the four anomalous incidents constructed in Experiment 1, generate examples of those anomalous incidents where the size of the base anomaly ranges from $AS = 2$ to 9.
- generate background test data stream using the same procedure as that laid out in Experiment 1.
- inject the constructed anomalous incidents into test streams.
- deploy the anomaly detector on the training and test data and vary the size of the detector window from 2 to 15.

8.3.4 Notes on detector deployment and scoring hits and misses

When a detector window slides over an anomaly, there will be instances where the detector window straddles elements of the background data and elements of the anomaly. Regardless of how the detector responds in such circumstances, the response is still influenced by the presence of the anomaly. Only when the detector window is completely clear of the entire anomaly (i.e., no elements that lie within the detector window belong to the anomalous sequence), can it be said that the detector is no longer influenced by, or responding to, the anomaly. In other words, as long as some part of the anomaly is seen by the sliding detector window, it can be argued that the detector's response was due to the presence of the anomaly. As a consequence of this observation, a decision was made to include, in the determination of hits and misses, the detector's response to instances where the detector window viewed some elements of the background data and some elements of the anomaly.

This decision gave rise to the idea of the incident span. The incident span includes the entire base anomaly, as well as the elements of the background data adjacent to the anomaly. More precisely, the incident span includes the $DW - 1$ elements of the background data adjacent to the anomaly on one side, the AS elements of the anomaly, and the $DW - 1$ events of the background data adjacent to the anomaly on the other side, (see Figure 8.3). The size of this span is therefore $AS + 2(DW - 1)$ elements, or it can be said that $AS + (DW - 1)$ sequences of size DW are contained within the incident span. Using the situation where only a single anomaly was introduced into each test stream, and *using 0 to signify normal and 1 to signify an anomaly*, a detector is described as:

- *blind* in the case where the anomaly signal, indicating complete normality, is registered for every sequence of the incident span. In other words, the most anomalous signal registered is the signal indicating a completely normal event;
- *weak* in the case where the maximum anomaly signal registered along the entire incident span is a user defined number that is not 1, but rather, a number close to 0: defining what can be accepted as a weak response;
- *strong* in the case where at least one “most anomalous” response (i.e, in this example, the number 1), was registered along the entire incident span. The reasoning behind this involves the effect of the detection threshold on the resulting hit and false alarm rate. Detection thresholds are often used to determine if a detector's response to an event deviates from normal sufficiently to call the event anomalous. When it is possible to set a variety of different detection thresholds, only the value that signifies the maximum possible deviation from normal behavior will consistently register as an alarm. Therefore, by defining the successful detection of given anomalies in terms of the greatest possible deviation from normal behavior for a detector, the result is independent of variations in the detection threshold. The result that a detector has successfully detected an anomaly will not change by, for example,

Anomaly incident	Detected?
FF_8 <i>ForeignSeqI_{DW}</i> <i>ForeignSeqB_{DW}</i>	Yes
FA_8 <i>CommonSeqI_{DW}</i> <i>CommonSeqB_{DW}</i>	Weakly
RA_8 <i>CommonSeqI_{DW}</i> <i>CommonSeqB_{DW}</i>	No

Table 8.2: The anomaly detection capability of the CSD detector with respect to the detection of foreign-symbol, foreign-sequence and rare-sequence anomalies.

“disappearing” or becoming a miss should the detection threshold be raised or lowered.

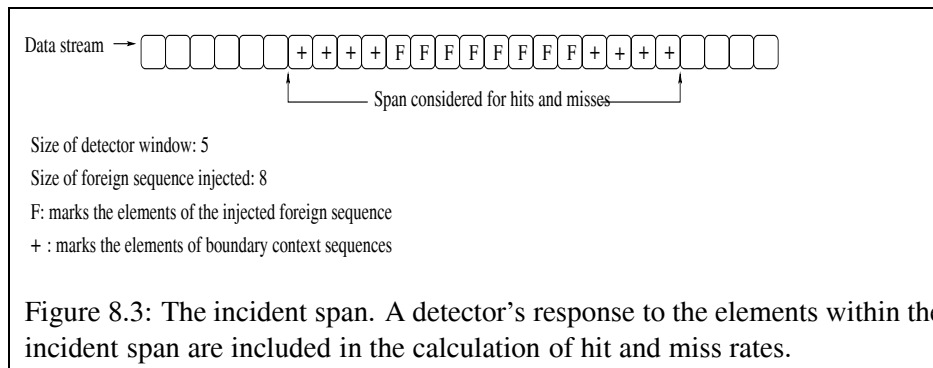
A hit or successful detection will be when the maximum response from the anomaly detector in the incident span is 1.0. A miss is when the maximum response from the anomaly detector in the incident span is 0.

8.4 Results

Experiment 1 identified whether the CSD was able to detect a member from each of the three basic anomaly classes, foreign-symbol anomaly, foreign-sequence anomaly, or rare-sequence anomaly, when the size of the detector window was set to be equal to the size of the base anomaly $DW = AS$. The results are presented in Table 8.2.

The CSD strongly detected the foreign-symbol anomaly, but only weakly detected the other two anomalies. Recall, that strongly detected means that at least one “most anomalous” response was registered along the entire incident span, and weakly detected means that the maximum anomaly signal registered along the entire incident span is a value close to the “most normal” response (see the section entitled “Notes on detector deployment and scoring hits and misses”).

For the CSD the most anomalous signal is 0, indicating a complete mismatch in every position between two sequences. The CSD strongly detected the foreign-symbol anomaly because every sequence in the incident span registered the most anomalous response of 0.



Detection weakness refers to the case where the most anomalous signal that was registered along the incident span was a value close to the most normal response for the detector. For the CSD this value is $Sim_{weak} = \sum_{i=1}^l i = \frac{2}{l(l-1)}$.

For the foreign-sequence anomaly, $FA_8 CommonSeqI_{DW} CommonSeqB_{DW}$, where $DW = AS$, the most anomalous response registered along the incident span was a single $Sim_{weak} = \sum_{i=1}^l i = \frac{2}{l(l-1)}$. This means that the CSD weakly detected the presence of this anomalies.

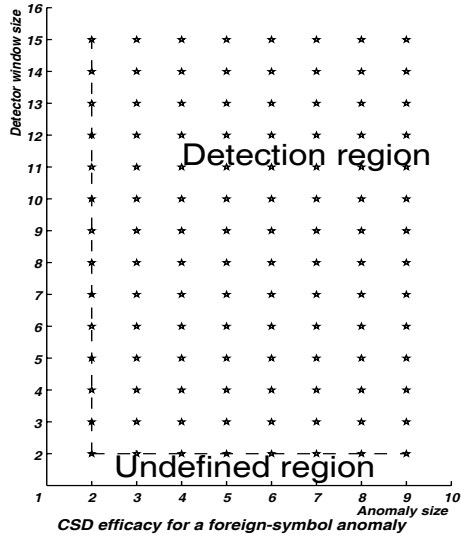
The CSD was completely blind to the same foreign-sequence anomaly, $FA_8 CommonSeqI_{DW} CommonSeqB_{DW}$ when $DW < AS$. The CSD was also completely blind to the rare-sequence anomaly, $RA_8 CommonSeqI_{DW} CommonSeqB_{DW}$ when $DW = AS$.

Experiment 2 aimed to identify the circumstances that would have enhanced or reduced the capabilities of the CSD with respect to the detection of each of the anomalies that it had successfully detected in Experiment 1. Experiment 2 focused on how the interaction between the size of the detector window and the size of the base anomaly affected detection performance. For each base anomaly size 2 to 9, the size of the detector window was varied between 2 to 15.

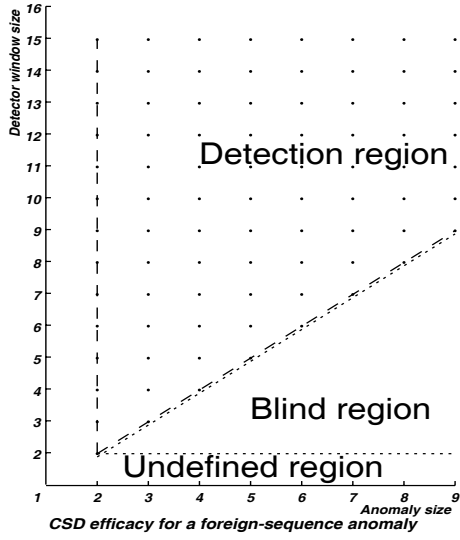
In Figure 8.4, Graph A shows the response of the CSD to the foreign-symbol anomaly, Graph B shows the response of the detector to the foreign-sequence anomaly and Graph C shows the response of the CSD to the rare-sequence anomaly. The stars indicate that the detector was capable of detecting the anomalous incident, i.e., at least one most anomalous signal was registered along the entire length of the incident span. Dots indicate weak detection capability, i.e., the value of the most anomalous signal registered along the length of the incident span was close to the value of a signal indicating a completely normal event.

8.5 Discussion

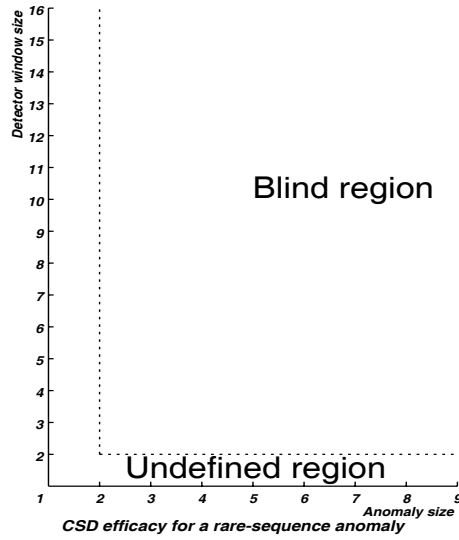
The results of the experiments uncovered an interesting characteristic of the CSD, attributable to the similarity metric that the detector employs. It appears that even when the CSD is able to view the entire foreign sequence, it is possible for the detector classify that foreign sequence as a normal sequence. As a point of comparison, stide from the work by Forrest et al. [6] under the same circumstances, i.e., when $DW = AS$, would have unquestionably detected the foreign sequence as an anomaly and set off an alarm. The CSD, however, may not. Some foreign sequences are more anomalous than other foreign sequences to the CSD.



A. Performance space for the CSD:
Foreign-symbol anomaly



B. Performance space for the CSD:
Foreign-sequence anomaly



C. Performance space for the CSD: Rare-sequence anomaly

Figure 8.4: The performance space for the CSD. The detector is completely blind to rare-sequence anomalies

In the experiments, the foreign-sequence anomalies presented to CSD were carefully constructed so that the entire set of context sequences for each foreign-sequence anomaly contained either only common sequences or rare sequences. Superficially, one would expect that when a foreign-sequence anomaly of size AS was presented to the CSD, and the detector window was set to view the entire foreign-sequence anomaly, i.e., $DW = AS$, the detector would respond either with the most anomalous signal, or close to the most anomalous signal; a foreign-sequence anomaly is after all a sequence of size AS that does not exist in the normal dictionary of sequences of size DW , where $DW = AS$.

The following questions arose as a result of the CSD's weak response to foreign-sequence anomalies despite the fact that the entire foreign-sequence anomaly was visible to the anomaly detector:

- Why was the response of the detector so weak for such an obviously anomalous phenomenon (i.e., a sequence that doesn't exist in the training data)?
- What are the characteristics of the foreign-sequence anomalies that elicit such a weak responses?
- What are the characteristics of the foreign-sequence anomalies that would elicit a stronger, or the strongest, response from the detector?

In response to the first question, it can be observed that the CSD's weak response to a foreign-sequence anomaly arises as a result of:

- the bias in the similarity metric toward matching adjacent elements;
- the existence of foreign sequences composed of rare or common context sequences whose "foreignness" is the result of a single element, and
- the position of that element within the sequence.

In order to answer the second question and to provide further explanation of the items listed above, note that it is possible to create a foreign-sequence anomaly with rare or common sequences such that only one position in the foreign-sequence anomaly differs from the most similar sequence in the normal dictionary. Recall that when a single test sequence is compared with every sequence in the normal dictionary, each comparison produces a similarity value. The final similarity value that is assigned to the test sequence is the one that resulted from the comparison between the test sequence and normal sequence most similar to the test sequence. In other words, the sequence in the dictionary that is most similar to the test sequence, where *similar* is determined by the similarity metric, provides the test sequence with its similarity value. This similarity value indicates how similar the test sequence is to the normal dictionary.

It is not possible for a foreign-sequence anomaly to have an identical sequence in the normal dictionary of sequences, but it is possible for a foreign-sequence anomaly to find a sequence in the normal dictionary that matches it in every position except one. Since the similarity metric for the CSD is biased in favor of

matching adjacent elements, when the single element involved in the mismatch is located at the very first or very last position of the foreign-sequence anomaly, the resulting similarity value will be close to the similarity value indicating complete normality. In other words, the similarity value increases for each successive match between a test sequence and the most similar sequence from the normal dictionary. The similarity value increases toward the limiting case of identical sequences, i.e., $Sim_{max} = \sum_{i=1}^{DW} i = \frac{2}{DW(DW+1)}$. However, if a single element is found to mismatch at the very last position of the sequence, the similarity value would be prevented from reaching the value which indicates identical sequences. Instead, the series of matching elements will result in a similarity value that comes close to the most normal response that the detector can produce, i.e., $Sim_{max} = \sum_{i=1}^{DW} i = \frac{2}{DW(DW+1)}$. A value that is close to the most normal response for the anomaly detector is not something that is expected as a response to a foreign sequence.

Figure 8.5 shows an example of the similarity calculation for a foreign-sequence anomaly with a mismatching element at the end of the sequence. This results in a similarity value that is closer to normal than the foreign-sequence anomaly containing a foreign element in the middle of the sequence 8.6.

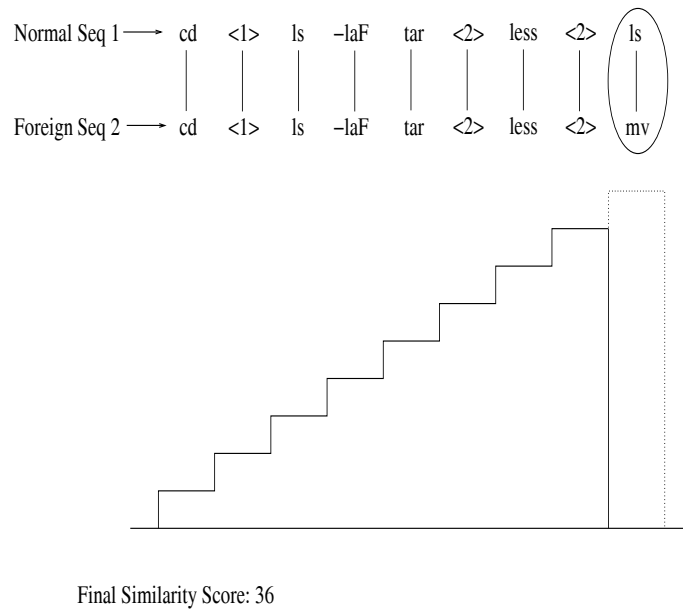


Figure 8.5: Similarity calculation between two size-9 sequences, where the foreign element lies at the end of the sequence. The oval shape marks the position of the foreign element in the foreign sequence. The step curve represents the weight contributed by each match. Dotted lines indicate the weight that would have been contributed if an exact match occurred in that position.

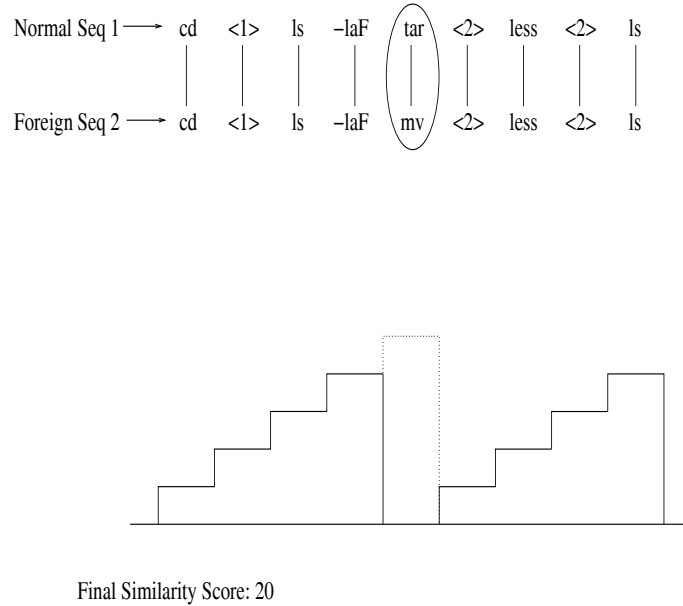


Figure 8.6: Similarity calculation between two size-9 sequences, where the foreign element lies in the middle of the sequence. The oval shape marks the position of the foreign element in the foreign sequence. The step curve represents the weight contributed by each match. Dotted lines indicate the weight that would be contributed if an exact match occurred in that position.

Note that this *most similar* foreign-sequence anomaly is a specific case of the minimal foreign-sequence anomaly. A most similar foreign-sequence anomaly is a minimal foreign-sequence anomaly where the element in each position of the anomaly matches the corresponding element in the most similar sequence found in the normal dictionary, except for the first or the last element.

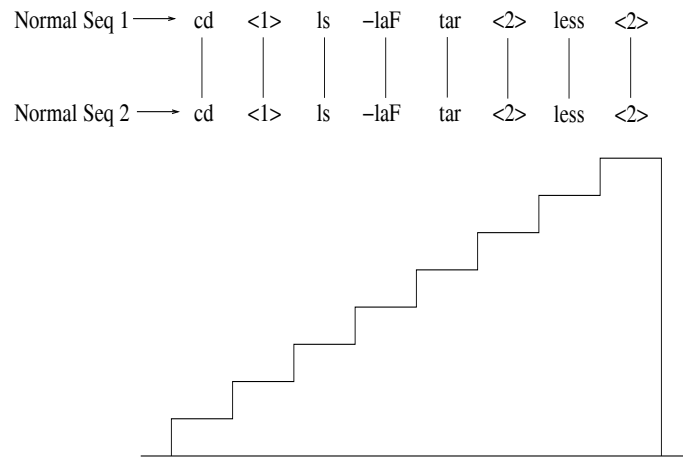
Since the similarity metric prevents the detector from producing the most anomalous signal, 0, in response to the presence of a completely foreign sequence, where $DW = AS$, the detection of the foreign sequence becomes totally reliant on the placement of the detection threshold. The CSD could completely miss the foreign sequence if the detection threshold were set such that sequences with similarity scores equal to and greater than $Sim_{weak} = \sum_{i=1}^l i = \frac{2}{l(l-1)}$ were considered normal.

To illustrate this point, Graph A in Figure 8.7 depicts the similarity calculation between two identical sequences of size 8. The similarity metric for these two identical sequences is $Sim_{max} = \sum_{i=1}^{DW} i = \frac{2}{DW(DW+1)} = 36$. This is the highest value that the similarity metric can produce for a detector window of size 8. However, in Graph B of Figure 8.7, the only difference between the two sequences of size 8 is the last element. The similarity metric for these two sequences is $Sim_{weak} = \sum_{i=1}^{DW} i = \frac{2}{DW(DW-1)} = 28$. If one of those two sequences in Graph B of Figure 8.7 is a foreign-sequence anomaly, then the slight dip from the similarity value from 30 to 28 is all that indicates the presence of that foreign sequence. This means that in order to detect the foreign-sequence anomaly, the detection threshold must be set close to the normal value of 30, i.e., 28, so that all similarity values less than or equal to 28 are considered anomalous. Furthermore, the slight dip that indicates the presence of such a foreign-sequence anomaly can only be expected to be concealed further when the smoothing window is applied to average the similarity values for the previous X sequences.

Like stide and t-stide [42], the CSD is impervious to data characteristics that are based on probabilistic characteristics, such as data regularity, since the algorithm does not employ any probabilistic principles. The strength of the detector appears to lie in the detection of foreign sequences. However, unlike stide and t-stide, the CSD relies heavily on the composition of foreign sequences. Even when the CSD is able to see the entire foreign sequence, some foreign sequences may appear more normal than others. In the case of the minimal foreign-sequence anomaly, all three detectors, stide, t-stide, and the CSD, need to set the detector-window size to match the size of the foreign-sequence anomaly in order to detect the anomaly. However, the composition of the foreign-sequence anomaly will only affect the strength of the resulting anomaly signal for the CSD, hence limiting the CSD's detection capabilities.

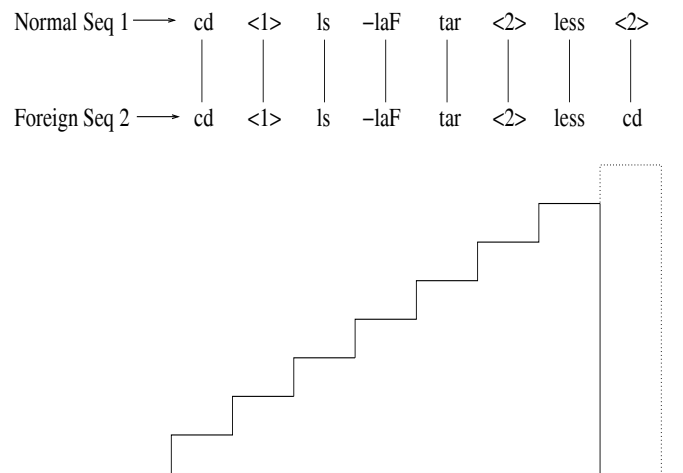
The CSD can be seen to rate foreign sequences. Given that the anomaly detector sees the entire foreign sequence, $DW = AS$, a foreign sequence can cause the CSD to produce:

- the most anomalous signal, 0, indicating the greatest degree of dissimilarity because every element in the foreign sequence is not identical to the corresponding element within the most similar sequence in the normal dictio-



Final Similarity Score: 36

A. Similarity calculation between two identical size-8 sequences. The step curve represents the weight contributed by each match.



Final Similarity Score: 28

B. Similarity calculation between two size-8 sequences that differ in the last element. The step curve represents the weight contributed by each match.

Figure 8.7: Describing a “most similar” foreign sequence.

nary to that foreign sequence. Since every element in the foreign-symbol sequence is alien to the alphabet of the training data, every element in the foreign-symbol sequence will incur a mismatch when it is compared with any sequence from the normal dictionary. These kinds of foreign sequences would elicit the strongest anomalous response from the CSD, the value 0;

- a semi-anomalous signal that is neither close to 0 nor close to the normal signal that results if two sequences were identical, i.e. $Sim_{max} = \sum_{i=1}^{DW} i = \frac{2}{DW(DW+1)}$. This could, for example, occur if two successive elements positioned at the very beginning or end of the foreign sequence were the only mismatches among the elements of the foreign sequence and the elements of the sequence in the normal dictionary that are most similar to the foreign sequence;
- a signal very close to normal, i.e, close to the normal signal $Sim_{max} = \sum_{i=1}^{DW} i = \frac{2}{DW(DW+1)}$ indicating two identical sequences. This could occur, for example, if only one element positioned at the very beginning or end of the foreign sequence was the only mismatch among the elements of the foreign sequence and the elements of the sequence in the normal dictionary that is most similar to the foreign sequence.

The CSD is completely blind to the presence of a foreign-sequence anomaly when the size of the detector window is set to be less than the size of the foreign-sequence anomaly. This means that the presence of a foreign sequence in the data stream will not always be reliably detected. In order for the CSD to reliably detect a foreign sequence with an anomaly signal value of 0, indicating the greatest deviation from normal behavior:

- the size of the detector window must be equal to or greater than the size of the foreign sequence, and
- there must not be a sequence in the normal dictionary whose elements match the elements in the foreign sequence at every position except for the first or the last position (see Graph B of Figure 8.7 for an example).

8.6 Conclusion - a summary of the lessons learned

From the study of the CSD and its operational limits, the following lists the lessons learnt that should be considered before the detector is deployed.

For the CSD as an anomaly detector

- The CSD detects only foreign sequences. However, there are circumstances where the detector will fail at detecting foreign sequences, for example, when $DW < AS$ for minimal foreign-sequence anomalies. In such a circumstance, the size of the foreign sequence must be less than or equal to the size of the sequences in the normal dictionary (the detector window size) in order to enable the CSD to detect the minimal foreign-sequence anomaly.

- All foreign sequences are not equally anomalous to the CSD. Some foreign sequences appear almost normal, while others produce the most anomalous response from the detector.
- The CSD detector is impervious to the effects of data irregularities that do not introduce foreign-symbol anomalies into the test data stream.

For the CSD as an anomaly-based intrusion detector

- It may not be possible to set the best detector-window length for the CSD a priori based only on normal data used to establish the normal dictionary. Since the best detector-window length, where the best length is one that will detect at least one anomaly in all the intrusive traces presented to the detector, is heavily dependent on the size and types of the various foreign sequences found in the test data. Foreign sequences are found only in test data; training data used to establish the normal dictionary will not contain foreign sequences.
- The larger the size of the detector window, the more likely the detector will detect all types of foreign sequences.

Task 9

Tool for elucidating the sequential dependencies in categorical audit-type data

Proem. Natural data have certain sequential dependencies that should be replicated in synthetic data. A tool is needed to ascertain the nature of such sequential dependencies.

9.1 Introduction

Natural data have certain sequential dependencies. Intuitively, one can observe this by noting that certain computer commands occur more frequently (or sometimes they occur only) after certain other commands. These sequential dependencies can be used to characterize data from certain environments. Extracting these characteristics provides researchers with a model that would enable them to synthesize data that mimics real-world environments. Synthetic data that closely mirrors real-world data is necessary because real-world data needed to tune or evaluate detection systems are typically difficult to obtain. A tool to ascertain the nature of sequential dependencies in natural data is described below. The source code for this tool is available in the Task 9 directory of the CD/DVD accompanying this report.

9.2 Description

The tupler is a Perl script that takes as input a stream of categorical data and produces as output a file containing a list of sequences that appear in the input data along with their corresponding frequencies.

The result of the tupler is a decomposition of a given sample of categorical data according to its constituent sequences. The size of these sequences is controlled by a parameter to the program. The frequencies with which these sequences occur can be used to produce synthetic data that mirror the characteristics of real-world environments.

The input to the tupler program is a stream of categorical data of the following form:

H I J A B C D E F G H C J A B C D

The program expects one element of categorical data per line. Although upper-case letters are used in this example, the input need not be restricted to the use of upper-case letters only. For example, a stream of strace data could be used:

open write close fork exit

or BSM, process accounting, tcpdump data, and so forth.

The tupler produces as output a stream of sequences and their corresponding frequencies. The following is an example of the tupler's output for a sequence size of 3.

```
500000
42403 C, D, E
42361 D, E, F
42285 G, H, I
42264 F, G, H
42245 B, C, D
42240 A, B, C
42188 E, F, G
42176 I, J, A
42173 J, A, B
42172 H, I, J
477 A, J, A
473 J, A, J
```

The first line of the output, 500000, denotes the size of the data which in this case means that there were 500000 categorical elements in the input data file.

The rest of the output lists all the size-3 sequences in the data, and their frequency of occurrence. To give an example, take the first line. The line 42403 C,D,E. This means that the elements D and E occur after the element C 42403 times. By simple calculation, the probability of D and E occurring after C can be obtained by:

$$\text{Prob(D,E occurring after C)} = 42403 / (500000 - (\text{Seq.size} - 1))$$

which becomes:

$$\text{Prob(D,E occurring after C)} = 42403 / 499998 = 0.0848$$

Sequential dependency refers to the probability that a given sequence occurs after a given categorical element. In the example above, the probability of seeing the sequence D,E after the element C is calculated to be 0.0848 using the given input data. In other words, there is, approximately, an 8.5% chance of seeing D,E after C.

9.3 Usage

“tupler” expects the Perl interpreter to be located in “/usr/local/bin/perl5”; if this is not the case, then insert the correct location of the Perl interpreter into the first line of the program.

Executing tupler in the following manner:

```
./tupler
```

will cause the program to list the method of usage, namely:

```
USAGE: tupler <window size> <filename>
```

where “window size” is the length of the sequence for analysis, and “filename” is the name of the file containing the categorical data to be analyzed.

“tupler” will print the results to standard output. If the results are required in a file then execute the script in the following manner:

```
./tupler 2 input_file > output_file
```

“output_file” is simply the name of the output file into which the results will be placed. Therefore the command above will result in tupler taking the file called “input_file” containing data in the format described above, extracting sequential dependencies of size 2 from the data and placing the results in “output file”.

Task 10

Tool for creating benchmark datasets that reflect intrinsic environmental structure

Proem. A given natural environment will yield data with intrinsic structure reflecting that environment. Replicating that structure requires a tool for incorporating that structure into synthetic benchmark data.

10.1 Introduction

Natural data is typically extremely difficult to obtain, particularly in the quantities required to evaluate detection systems. As a consequence, synthetic data is often used to evaluate detection systems.

It has been observed that a given natural environment will yield data with intrinsic structure reflecting that environment. It is important to capture this “natural” structure in order to generate synthetic benchmark datasets that will more closely resemble real-world data. Testing detection systems using synthetic data that closely replicates the “natural” structure in real-world data will provide a better understanding of how such detectors will operate in real-world environments.

The following is a description of a tool called “reconstruct” that captures the intrinsic structure characteristic of a given natural environment. The software on the accompanying CD/DVD is under Task 10.

10.2 Description

The “reconstruct” program takes categorical data as input from a data file and produces a Markov transition matrix which captures the probabilistic structure within the input data. This structure reflects the probabilistic characteristics of the given data environment and can be used to synthesize benchmark data that incorporates or replicates the same characteristics that was found in the given data environment.

The input to the reconstruct program is a stream of categorical data of the following form:

A A B F G H I J A B C D E F G H C J A B C D E

The program expects one element of categorical data per line.

Reconstruct produces as output a Markov transition matrix. The following is an example of the output from reconstruct:

	A	B	C	D
A	0.03	0.91	0.03	0.03
B	0.03	0.03	0.91	0.03
C	0.03	0.03	0.03	0.91
D	0.91	0.03	0.03	0.03

The first line lists the individual categorical elements found in the input file (no repetitions). In this case, the elements A, B, C, and D were found in the input file. The first column repeats these elements, and the numbers of the matrix denote the probabilities associated with the occurrence of each element. For example, observe the first and second line of the matrix:

	A	B	C	D
A	0.03	0.91	0.03	0.03

This can be read as follows:

- The element A (in the top first line) follows the element A (in the first column) with a probability of 0.03.
- The element B (in the top first line) follows the element A (in the first column) with a probability of 0.91.
- The element C (in the top first line) follows the element A (in the first column) with a probability of 0.03.
- The element D (in the top first line) follows the element A (in the first column) with a probability of 0.03.

The rest of the output is read in the same way. Using this transition matrix, a stream of synthetic data reflecting these probabilities obtained from the input data sample can be generated.

10.3 Usage

The executable for reconstruct.c, the file named “reconstruct”, currently works on Linux. To use reconstruct on other platforms, recompile reconstruct.c, e.g., execute:

```
gcc -o file.execute reconstruct.c
```

where “file.execute” is the name of the executable. In this documentation the executable for reconstruct.c will be referred to as reconstruct.

Executing reconstruct in the following manner:

```
./reconstruct
```

will cause the program to list the method of usage, namely:

```
USAGE: reconstruct <datafile> <outputfile>
```

where “datafile” is the name of the file containing the categorical data, one element per line.

“outputfile” is simply the name of the output file into which the resulting transition matrix will be placed. Therefore:

```
./reconstruct mary john
```

will result in reconstruct taking the file called “mary” containing data in the format described above, extracting the probabilistic characteristics from the data in “mary” and placing the resulting transition matrix into the file named “john”.

10.3.1 recon

Description

“recon” is a Perl wrapper for reconstruct.c . The function of the wrapper is to cut a datafile by a given percentage, call reconstruct.c on the that cut subset of the original datafile to extract the resulting transition matrix.

This wrapper is aimed for use in situations where only a smaller subsection of the natural data environment is targeted for analysis. This can be useful in cases where the size of the natural data exceeds the resources used to analyze it.

Usage

“recon” expects the Perl interpreter to be located in “/usr/local/bin/perl5”, if this is not the case, then insert the correct location of the Perl interpreter into the first line of the wrapper.

Executing recon in the following manner:

```
./recon
```

will cause the program to list the method of usage, namely:

```
USAGE: recon <datafile> <percent of datafile.Omit '%'> <outputfile>
```

where “datafile” is the name of the file containing the categorical data, one element per line, as described above in the “Usage” section of reconstruct.c .

“percentage of datafile required” is the amount of the data in “datafile” that is to be used to produce the transition matrix. So 50 would mean that the first 50%

of the datafile will be used, 20 would mean that the first 20% of the datafile will be used and so forth. Percentage signs are omitted.

“outputfile” is simply the name of the output file into which the resulting transition matrix will be placed. Therefore:

```
./recon mary 50 john
```

will result in recon taking the file called “mary” containing data in the format described in the “Usage” section of reconstruct.c, extracting the first 50% of the data in the file called “mary”, executing reconstruct to obtain the probabilistic characteristics from 50% of the data in “mary” and finally placing the resulting transition matrix into the file named “john”.

Task 11

Tool for generating benchmark datasets containing specified regularity

Proem. Detectors can be sensitive to regularity in data. A tool is needed for generating a series of benchmark data sets that contains specific regularities spanning a region of sensitivity, thus creating an opportunity for performing a sensitivity analysis of a detector.

11.1 Introduction

Detectors can be sensitive to regularity in data. Intuitively, a detector can be expected to have a more difficult time detecting anomalies in environments consisting of random behavior as opposed to environments exhibiting regular behavior. As a consequence, the regularity of a given data environment can be expected to be an influential factor in detector effectiveness.

This chapter describes a tool for generating a series of benchmark data sets that contains specific regularities spanning a region of sensitivity between random behavior and extremely predictable behavior. These data-sets allow for sensitivity analyses to be performed on anomaly detectors. The tool that generates these benchmark data sets is called Syngen; the source code for this tool is available in the Task 11 directory of the accompanying CD/DVD.

Note that these data-sets were designed to test the sensitivity of a detector over a span of data environments that may or may not reflect any particular natural environment. If the desire is to evaluate the performance of a detector on data that is typical of a given natural environment, then a sample of data from that natural environment must first be procured and the program “reconstruct” (from task10) must be executed in order to determine the regularity of that sample environment so that data typical of that environment can subsequently be generated.

11.2 Description

Syngen generates synthetic data of given regularities. It takes as input a configuration file and produces 5 files as output. These files are listed and explained below.

- training-data file - contains training data of a specified regularity.
- test-data file - contains test data of a specified regularity with no injected anomalies.
- injected-test-data file - contains data from the test-data file and injected anomalies.
- key file - contains the exact location of the injected anomalies (ground-truth).
- anomaly file - a file listing the anomalies that were injected into the “injected-test-data file”.

The configuration file that Syngen takes as input contains the various parameters required by the program, e.g., the size of the data-set to be generated, the names of files in which to place the output, and so forth. It also serves as a documentation of what was accomplished in one run of the program. Following is an example of the configuration file:

```
#comments
1 size of dataset: 500000
2 regularity value (0, 0.1, 0.2, ..., 0.8, or, 1.0): 0.2
3 name of training file: trainfile
4 seed for training file generation: 16257392
5 name of test file: testfile
6 seed for test file generation: 6246556
7 name of injected test file: injected_testfile
8 name of key file: keyfile
9 class of anomaly: F
#class of anomalies injected will be rare sequences of size 6.
#Rare defined as relative freq. of less than 0.05
10 size of anomaly: 6
11 number of anomalies to be injected: 250
12 anomalyfile: Anomalies
13 alphabetsize of data: 6
14 spacing between anomalies: 10
```

Syngen expects the “#” symbol to indicate a comment line. The first line of the configuration file determines the desired size of the training and test data-sets. In the example above the training and test data-sets are set to be 500000 elements long. The subsequent paragraphs will explain in detail lines 2 and 9. All the other lines are explained in the configuration file itself, for example, line 5 is the name of the testfile that Syngen expects to place the generated test data into, line 6 is the seed for the random number generator, line 7 is the name of the injected test-file that Syngen expects to place the anomaly-injected test-data into and so forth.

Line 2 determines the regularity of the generated data. There are 11 matrix files in this director labeled from 0.0 to 1.0 in increments of 0.1. These matrix files contain transition tables of given regularities. A transition table of a given regularity will be used to generate data of that same regularity, for example, the file labeled “0.7” contains a transition matrix of regularity 0.7 that will generate data with a regularity of 0.7. The matrix with the regularity of 0.0 will generate the most regular (predictable) data, and the matrix with a regularity of 1.0 will generate the most random data. Following is an example of a transition matrix, this example displays the transition matrix for the extremely regular data, i.e., regularity 0.0.

```

6
A A 0.000000
A B 1.000000
A C 0.000000
A D 0.000000
A E 0.000000
A F 0.000000
END
B A 0.000000
B B 0.000000
B C 1.000000
B D 0.000000
B E 0.000000
B F 0.000000
END
C A 0.000000
C B 0.000000
C C 0.000000
C D 1.000000
C E 0.000000
C F 0.000000
END
D A 0.000000
D B 0.000000
D C 0.000000
D D 0.000000
D E 1.000000
D F 0.000000
END
E A 0.000000
E B 0.000000
E C 0.000000
E D 0.000000
E E 0.000000
E F 1.000000
END

```

```

F A 1.000000
F B 0.000000
F C 0.000000
F D 0.000000
F E 0.000000
F F 0.000000
END

```

The first number, 6, denotes the alphabet size of the data-set. The entries can be read in the following way, for the second line

```
A A 0.000000,
```

the probability that the letter “A” follows a previous letter “A” is 0.000000. For the third line,

```
A B 1.000000,
```

the probability that the letter “B” follows a previous letter “A” is 1.000000, and so forth. The word “END” is used as a separator.

Line 9 denotes the class of anomaly. There are 3 classes of anomalies that Syngen can inject into the test data-stream, rare sequence, foreign sequences and foreign symbols. Rare sequence anomalies are denoted by an “R”, foreign sequences by an “F” and foreign symbols by an “X”.

Rare sequences require an added value, namely the definition of rare. Rare is defined by the relative frequency of a sequence, “anomaly_syn” expects the request for rare sequence anomalies to be accompanied by the relative frequency by which “rare” is defined. For example, R-0.04, means that rare sequence anomalies are requested where rare is defined as sequences that occur with less than a relative frequency of 0.04 (i.e., sequences that occur with a frequency of less than 4% in the training data). In another example, if the definition of rare is a sequence with a relative frequency of 0.09, then anomaly_syn should be called in the following way:

```
./anomaly_syn 6 R-0.09 trainfile 6 Anomalies
```

where the first parameter “6” denotes the size of the anomaly, the second parameter denotes the type of anomaly - R means rare sequence and 0.09 means sequences are rare if they have a relative frequency of less than 0.09, the third parameter is the name of the file containing the training data, the fourth parameter is the alphabet size and the last parameter is the name of the output file that will contain the list of rare sequence anomalies.

If the anomaly type requested cannot be found given the training data, Syngen will display a warning that the anomaly type cannot be found and exit, leaving the generated training and test files without the injected test-file, key file or anomaly file, all of which require the presence of anomalies to exist. Note that the most regular transition matrix will not produce data that will possess rare sequences. This is because every categorical element of the regular will have the same chance of occurring as every other element and hence there are no elements that will occur less frequently than others resulting in the absence of rare elements.

11.3 Running Syngen

To run Syngen, type in:

```
./Syngen CONFIGURATION
```

CONFIGURATION is the name of the configuration file described above. The steps itemized below are the processes that Syngen will perform.

- Generate training data of a specified regularity using a Markov transition matrix of the same specified regularity.
- Generate background test data of the same regularity as the training data.
- Synthesize anomalies using the training data.
- Inject synthesized anomalies into the background test-data to produce the injected-test-data file.

The entire process produces 5 outputs, the training-data file, the test-data file, the injected-test-data file, the key file and the anomaly file.

Below is an example of the data in the training file:

D
E
F
A
B
C
D
E
F
A
B
C
D
E
F
A
B
C
D
E
F
A

The test and injected-test-data files will as given in the example above, one categorical element per line. The injected-test-data file differs from the test-data

file in that the injected test data will contain injected anomalies while the test data will not.

The following is an example of the key file.

```
1692 1697 D D A C F C
2724 2729 D D A C F C
4714 4719 D D A C F C
5808 5813 D D A C F C
7819 7824 D D A C F C
11310 11315 D D A C F C
13004 13009 D D A C F C
15008 15013 D D A C F C
16539 16544 D D A C F C
17293 17298 D D A C F C
19213 19218 D D A C F C
21139 21144 D D A C F C
24672 24677 D D A C F C
24957 24962 D D A C F C
```

The first line is read as such: the anomalous sequence

D D A C F C

can be found beginning on line 1692 and ending on line 1697 in the injected-test-data file.

The following is an example of the anomaly file.

```
6
D D A C F C
```

The first line denotes the size of the anomalous sequence injected into the test data, in this case 6.

D D A C F C

is the anomaly of the requested type (e.g., rare or foreign sequence, or foreign symbol) that was found in the training data and that was injected into the background test data.

Task 12

Benchmark datasets from enterprise environments

This task was de-scoped and omitted due to sponsor exigencies.

Task 13

Series of benchmark datasets containing well-specified ground-truth

Proem. Accurate ground truth is a necessary and rare commodity in testing detection algorithms. Data with injected intrusive activity can be generated to include a table showing intrusion locations highly accurately.

13.1 Introduction

One of the most crucial ingredients for evaluating anomaly detectors is well-defined data. Without a clear understanding of the data, where ground-truth involves not only knowing the characteristics of the background data, but also the location and characteristics of the anomalies, it is impossible to acquire dependable and unambiguous results from an evaluation process that aims to map a given detector's operational behavior and limits.

This document describes a series of benchmark data-sets for evaluating anomaly detectors. These data-sets have well-specified structure in terms of predictability or regularity, and ground-truth in terms of the location and characteristics of the anomalies contained within. The data-sets are tailored for anomaly detectors, and is aimed at evaluation processes that establish a given detector's operational efficacy. These data sets are available in the Task 13 directory of the accompanying CD/DVD.

13.2 Description

The benchmark data-sets are grouped into 3 directories:

- train - this directory contains training-data files.
- test - this directory contains test-data files.

- key - this directory contains files that list the exact location of the injected anomalies (ground-truth).

1. In the directory named “train” there are 11 training files of differing regularities. They are labeled:

```
0.0_train
0.1_train
0.2_train
0.3_train
0.4_train
0.5_train
0.6_train
0.7_train
0.8_train
0.9_train
1.0_train
```

The name of the file is composed of a number and the word “train” to denote that the file contains training data. The number specifies the degree of regularity that is contained in the training data. 0.0_train contains the most regular data, and 1.0_train contains the most irregular. The regularity of the training data steps up in carefully calibrated steps of 0.1 between 0.0 and 1.0. The following is an example of the contents of a file containing training data:

```
F G A H A B C D E F H C D A B C B C B
```

The format is simply a single categorical element per line.

2. In the directory named “test” there are 10 training files of differing regularities. They are labeled:

```
0.1_test
0.2_test
0.3_test
0.4_test
0.5_test
0.6_test
0.7_test
0.8_test
0.9_test
1.0_test
```

Note that the test files correspond to the training files via the number in the file-name, e.g., a detector trained on the training file “0.0_train” should test using the test data found in the file “0.0_test”. This is important because data regularities are carefully calibrated to match between training and test in order to avoid confounding results.

Also note that the test file “0.0_test” is missing, this is a consequence of the extremely regular nature of the data in 0.0_train and the type of anomaly injected into

the test files. The anomalies injected into the test files are rare sequence anomalies. These are among the more difficult anomalies to detect, detectors such as stide, for example, would not be able to identify rare sequence anomalies at all. Anomalies are created using training data as the base. For example, sequences foreign to the training data are used as foreign sequence anomalies, and sequences that are rare in the training data are used as rare sequence anomalies. In the case of the training data in file “0.0_train”, rare sequences were unavailable for constructing anomalies because “0.0_train” contains the most regular data, this means that every sequence occurs as frequently as every other. Under these circumstances there were no rare sequences for anomalies and as a consequence no “0.0_test” file.

The following is an example of the contents of a file containing test data, the format is identical to that of the training data:

```
A
B
C
A
A
A
B
B
C
D
E
C
B
A
```

3. In the directory named ”key” there are 10 key-files corresponding to the 10 test files of differing regularities. They are labeled:

```
0.1_key
0.2_key
0.3_key
0.4_key
0.5_key
0.6_key
0.7_key
0.8_key
0.9_key
1.0_key
```

where “0.2_key” is the key-file for the test file “0.2_test”, “0.3_key” is the key-file for “0.3_test”, and so forth. The following is an example of the contents of a key-file:

1126 1129
2156 2159
4727 4730
5831 5834
6831 6834
9438 9441
10536 10539
12548 12551

The first number is the line number where the anomaly begins and the second number is the line number where the anomaly ends. Therefore in the first line of the example above, an anomaly of size 4 was injected into the test file at line 1126 ending on line 1129. The second anomaly begins at 2156, and ends at 2159, and so forth.

Task 14

Benchmark dataset specifically tailored to address the insider-threat and profiling problems

Proem. Most insider (masquerader) data are injected not with insider activity, but with activity from other users intended to imitate insiders. Because these data sets are unrealistic, data specific to the insider problem, specifically tailored to stress a detection algorithm, are needed.

14.1 The dataset

The insider-threat benchmark data set is available in the Task 14 directory of the accompanying CD/DVD. This dataset contains synthetic user and insider data collected from 13,310 distinct simulated environments. An environment consists of two or more legitimate users and an insider each typing commands at a UNIX command line. The commands that each user types are recorded. The commands themselves are sanitized by substituting a unique number to represent each distinct command typed by the users. (E.g., the command 'ls' might be converted to the number 13).

Each environment contains training and test data. The training data is divided into data from one "self" user and data from one or more "nonself" users. Each user's training data consists of 1000 commands, chosen depending on the selected environmental characteristics. The test data consists of 10 commands from an insider, injected after the end of the "self" user's training data. Which 10 commands are chosen by the insider depends on the selected environmental characteristics. In this carefully controlled environment, the insider only uses two types of commands. One is a command that also appears in the self or nonself training data. The other is a command which has never been seen before. We call these two commands the "seen before" command and the "never before seen" command.

14.1.1 Environmental characteristics

The number of users and the types of commands issued by each legitimate user and the insider depend upon the characteristics of the particular environment. Data for each environment is generated with the following characteristics specified beforehand:

NSCount: The *nonself count* is the number of “nonself” users in the environment. There is always a “self” user and then between 1 and 5 “nonself” users. The training data of the nonself users are available to an insider-threat profiler to build a model of nonself (to contrast with its model of self). In this dataset, the nonself count ranges between 1 and 10.

SVal: The *self value* indicates the degree to which the insider’s “seen before” command resembles the commands of the self user. More specifically, a value of 1 means that the “seen before” command constitutes 0.1% of the self training data. A value of 10 means that the “seen before” command constitutes 1.0% of the self training data. In this dataset, the self value ranges between 1 and 10.

NSVal: The *nonself value*, a dual of the self value, indicates the degree to which the insider’s “seen before” command resembles the commands of the nonself user. A value of 1 means that the “seen before” command constitutes 0.1% of the nonself training data and a value of 10 means that it constitutes 1.0% of the nonself training data. In this dataset, the nonself value ranges between 1 and 10.

NBSC: The *never before seen command count* is the number of “never before seen” commands in the insider’s block of commands. A value of 0 means that only the “seen before” command appears in the insider’s block. A value of 5 means that the “seen before” command appears 5 times and the “never before seen” command also appears 5 times. A value of 10 means that only the “never before seen” command appears. In this dataset, the never before seen command count ranges between 0 and 10.

Data with different environmental characteristics are kept in separate folders.

14.1.2 Top-level folder

The folders are arranged in a hierarchy four levels deep. At the top level, there are 10 folders:

NSCount=1, NSCount=2, ... NSCount=10.

The “NSCount” label means “nonself count” and the value corresponds to the number of nonself users in the dataset. (A more detailed description of what these values mean, with respect to the character of the training and testing data, was provided in chapter 4.)

14.1.3 Second-level folder

In each top-level folder, there are 11 second-level folders:

$$SVal=0, SVal=1, \dots SVal=10.$$

The “SVal” label means “self value” and the value indicates frequency with which the insider’s “seen before” command appears in the self user’s training data.

14.1.4 Third-level folder

In each second-level folder, there are 11 third-level folders:

$$NSVal=0, NSVal=1, \dots NSVal=10.$$

The “NSVal” label means “nonself value” and the value indicates the frequency with which the insider’s “seen before” command appears in the nonself user’s training data.

14.1.5 Fourth-level folder

In each third-level folder, there are 11 fourth-level folders:

$$NBSC=0, NBSC=1, \dots NBSC=10.$$

The “NBSC” label means “never-before-seen command count” and the value corresponds to the number of times the “never before seen” command appears in the insider’s block of commands.

14.1.6 Data files

In each fourth-level folder, there are between 2 and 11 data files. If the top-level folder has an “NSCount” value of N , then there are $(N + 1)$ data files in this directory:

$$User1, User2, \dots UserN.$$

The user data labeled “User1” is the “self” data. The data with other labels is the “nonself” data.

Each data file contains 101 lines. Each line corresponds to a block of 10 commands. The following is an example line extracted from one of the data files:

$$c[4], c[5], c[6], c[7], c[8], c[9], c[9], c[10], c[10], c[15]$$

The number in brackets represents the command that was typed. In this case, the commands represented by the numbers 4, 5, 6, 7, 8, and 15 were typed once.

The commands represented by the numbers 9 and 10 were each typed twice. The letter “c” is simply a prefix identifying the number as representing a command.

The first 100 lines in the “User1” data file are the 1000 commands of self training data. The first 100 lines of the other data files are the 1000 commands (per nonself user) of nonself training data. The 101-st line in the “User1” data file is the insider’s block.

14.1.7 The environmental description file

Accompanying the data files in each fourth-level folder is a file describing the particular environment, named `ExpDesc`. The contents of the file are human readable. It contains the specific settings for each of the environmental characteristics. For example, the `ExpDesc` file in the

`NSCount=4/SVal=8/NSVal=3/NBSC=3`

folder-path contains the following description:

```
NSCount: 4
SVal: 8
NSVal: 3
NBSC: 3
```

This file is used to confirm that the synthetic data actually generated corresponds to the settings which the data was intended to match.

14.1.8 The command statistics file

Also, accompanying the data files in each fourth-level folder is also a file describing the number of times each command appears in the self and nonself training data. Each line of this file contains three numbers. For example, the following line is excerpted from one of these files:

`8,7 95`

The last number (95) is the number representing the command. The first number (8) is the number of times that command appears in the self training data. The second number (7) is the number of times that command appears in the nonself training data. This file is also used to confirm that the synthetic data conforms to the specified environmental characteristics.

14.1.9 The answer key file

Also, accompanying these data files in each fourth-level folder is an answer key file, named `AnswerKey`. This file is suitable for use with the naive Bayes masquerade detector. This file has 101 lines, corresponding to the 101 lines of the self data file. The first 100 lines contain a single zero (0) indicating the line is self training data. The final line contains a single (1) indicating the line is insider data.

14.1.10 The config file

Finally, accompanying these data files is a configuration file named `Config`. This configuration file is suitable for use with the naive Bayes masquerade detector. It is generated to allow the masquerade detector to be automatically run in each environment once the data for the environment has been generated and vetted. It is included in this dataset for completeness.

Task 15

Data-gathering and generating methodologies

This task was de-scoped and omitted due to sponsor exigencies.

Task 16

Guidance in the use of benchmark datasets for anomaly detection

This task was de-scoped and omitted due to sponsor exigencies.

Task 17

Assessment: Assessing a detector in the insider-threat environment

Proem. The way a detector is evaluated can make a huge difference in how its performance is perceived. A detector should be assessed not only in terms of its success, but also its failures, as shown in an error analysis that reveals *why* the detector sometimes misperformed. This chapter illustrates a careful and thorough analysis of a masquerade detector.

A masquerade attack, in which one user impersonates another, may be one of the most serious forms of computer abuse. Automatic discovery of masqueraders is sometimes undertaken by detecting significant departures from normal user behavior, as represented by a user profile formed from system audit data.

A major obstacle for this type of research is the difficulty in obtaining such system audit data, largely due to privacy concerns. An immense contribution in this regard has been made by Schonlau et al., who have made available UNIX command-line data from 50+ users collected over a number of months. Most of the research in this area has made use of this dataset, so this chapter takes as its point of departure the Schonlau et al. dataset and a recent series of experiments with this data framed by the same researchers [34].

In extending that work with a new classification algorithm, a 56% improvement in masquerade detection was achieved at a corresponding false-alarm rate of 1.3%. In addition, encouraging results were obtained at a more realistic sequence length of 10 commands (as opposed to sequences of 100 commands used by Schonlau et al.). A detailed error analysis, based on an alternative configuration of the same data, reveals a serious flaw in this type of data which hinders masquerade detection and indicates some steps that need to be taken to improve future results. The error analysis also demonstrates the insights that can be gained by inspecting decision errors, instead of concentrating only on decision successes.

17.1 Introduction

Colloquially, the masquerade problem can be described with the following scenario. A legitimate user takes a coffee break, leaving his/her terminal open and

logged in. During the user's brief absence, an interloper assumes control of the keyboard, and enters commands, taking advantage of the legitimate user's privileges and access to programs and data. The interloper's commands may comprise read or write access to private data, acquisition of system privileges, installation of malicious software, etc. Because the interloper is impersonating a legitimate user, s/he is commonly known as a masquerader. The term may also be extended to encompass the case of abuse of legitimate privileges, that is, the case in which a user "masquerades" as him/herself; e.g., an insider.

The assumption underlying the common approach to detection of such illegitimate activity is that the masquerader's behavior (including the illegitimate behavior of a legitimate user) will deviate from the historical behavior of the legitimate user. User profiles can be constructed from monitored system-log or accounting-log data. Examples of the kinds of information that can be derived from these (and other) logs are: time of login, physical location of login, duration of user session, cumulative CPU time, particular programs executed, names of files accessed, and so forth [21]. This chapter is focused on the latter attributes, because recent research in masquerade detection has been based largely on the Schonlau et al. dataset, which consists exclusively of the commands issued by users running the UNIX operating system, without arguments or other elements of the command line.

Masquerading can be a serious threat to the security of computer systems and the computational infrastructure. A well-known instance of masquerader/insider activity is the case of Robert P. Hanssen, the FBI mole who allegedly used agency computers to ferret out information later sold to his co-conspirators [19]. Hanssen's status as a user was legitimate, but his behavior was certainly not. More than 60% of companies surveyed in 1996 reported such misuse, at enormous cost. Attacks on national security are clearly even more of a concern than economic losses from attacks on business.

There have been several attempts to tackle the problem of detecting masqueraders. A nice collection of such work, in which a number of masquerade-detection techniques were applied to the same data set, is presented by Schonlau and his colleagues [34]. In terms of minimizing false alarms, the best result achieved in that work used a metric based on the uniqueness of commands to a user, obtaining a 39.4% hit rate with a corresponding 1.4% false alarm rate. In terms of detecting masqueraders, the best results reported in the Schonlau et al. work were for a Bayes one-step Markov model, with a 69.3% hit rate and a 6.7% false alarm rate.

This chapter takes the work of Schonlau et al. as a point of departure. It uses the data provided by Schonlau et al., and demonstrates a new technique for masquerade detection which yields a 56% improvement in correct detection at the lowest false alarm rate reported in the literature so far. Experiments were also conducted using a more realistic sequence length of just 10 commands, as opposed to the 100 used by Schonlau et al.; encouraging results were obtained. In addition, a thorough analysis of the errors made by the detector on an alternative configuration of the same data is provided. This error analysis exposes a serious impairment of the Schonlau et al. data, providing insight into what allows a masquerader to evade detection.

17.2 Problem and approach

Two objectives dominated this study: to determine whether or not a new classifier could improve upon the detection rates reported in previous work; and to provide a detailed examination of whatever classification errors occur in the study. An error analysis facilitates more learning than a simple report of classification results, because the errors and their causes will show what needs to be done to effect improvements in the future.

The problem is to detect illegitimate user activity. The assumption is that illegitimate activity will manifest as a deviation from the historical behavior of the user with respect to some attribute. The most recent research in masquerade detection has been based on the Schonlau et al. dataset, which consists exclusively of UNIX user commands [34]. These are the data used in this study. Thus, the attribute under consideration is command usage, and the task is to classify sequences of UNIX commands as “self” (issued by the authorized user), or “non-self” (issued by a masquerader). This task is reminiscent of text-classification, in which a document, seen as a collection of words, must be assigned to the class of politics or sports, for example. A classification algorithm known as Naive Bayes has a history of good performance in text-classification [29], and is employed as the detection algorithm in this work. Details about the detection algorithm are supplied in Section 17.4.

The following section discusses related work, after which a description is provided for the algorithm employed in the current study. Thereafter, details of data and experiments are given, followed by results, error analysis and discussion.

17.3 Related Work

Masquerade data are extremely hard to obtain, due to concerns about user privacy and corporate confidentiality. Recently, Schonlau and his colleagues made a major contribution by publishing a large collection of UNIX user command data. This dataset formed the basis for experimentation by several researchers using a number of masquerade-detection techniques, the results of which were published as a compendium in [34].

These authors used UNIX command data from 50 users, injected with normal data from other users, to model a masquerade attack. Data for each user comprised 15,000 commands. The first 5,000 commands constituted training data (i.e., were free of injected commands); the remaining 10,000 commands were probabilistically injected with commands from other users. The idea was to discriminate sequences of 100 commands typed by the target user from injected sequences of 100 commands taken from other users. Further details regarding the data can be found in Section 17.5.

The review paper by Schonlau et al. [34] compared the performance of six masquerade-detection algorithms (some new, others drawn from the computer science literature) on the data described above. Researchers sought to target a false alarm rate of 1%. All methods had relatively low hit rates (39.4% - 69.3%) and high false alarm rates (1.4% - 6.7%). The results were compared using both cluster analysis and ROC curves, revealing that no single method completely dominated

any other. An overview of the various masquerade detectors used by Schonlau et al. is provided below.

Uniqueness. This approach, due to Schonlau and Theus [35], is based on ideas about command frequency. It postulates that commands not seen in the training data are indicative of a masquerade attempt and that the fewer the users employing a command, the more indicative that command may be of a masquerade. Uniqueness was a poor performer in terms of detecting masqueraders (39.4%), but it was the only method able to approach the target false alarm rate of 1% (1.4%) [34].

Bayes 1-Step Markov. This detector is based on single-step transitions between commands, and is due to DuMouchel [5]. The detector determines whether or not observed transition probabilities are consistent with historical probabilities. This technique was the best performer in terms of correct detections (69.3%), but it failed to get close (6.7%) to the desired false alarm rate.

Hybrid Multi-Step Markov. This method is based on Markov chains, and is due to Ju and Vardi [12]. The implementation of this model in [34] actually toggled between a Markov model and a simple independence model, depending on the proportion of commands in the test data that had not been observed in the training data. The performance of this method was among the best of the methods tested (49.3% hits, 3.2% false alarms).

Compression. The idea behind the compression approach is that new data from the same user should compress at about the same ratio as old data from that user, whilst data from a masquerading user will compress at a different ratio and thereby be distinguished from the legitimate user. (In fact, the experiment described uses a one-sided test, assuming only that masquerader data will be harder to compress than legitimate user data.) This idea is credited to Karr and Schonlau in [34]. Compression was the worst performer of the methods tested, with 34.2% hits and 5.0% false alarms.

IPAM. This detector (incremental probabilistic action modeling) is based on single-step command transition probabilities, estimated from training data, and was developed by Davison and Hirsh [2] for their work in predicting sequences of user actions. The performance of IPAM was poor (41.1% hits, 2.7% false alarms).

Sequence-Match. This approach is based on the early work of Lane and Brodley, refined in [16]. A similarity match is computed between a user's most recent commands and a corresponding user profile. As implemented on the Schonlau et al. data, this method was a poor performer (36.8% hits, 3.7% false alarms).

17.4 The Naive Bayes Algorithm

This section provides a rationale for the use of a Naive Bayes classification algorithm in masquerade detection, and describes the details of the algorithm.

17.4.1 Why use Naive Bayes for masquerade detection?

Naive Bayes classifiers (also known as simple Bayes classifiers) have a long history of use in pattern recognition and text classification. They have a number of attractive attributes: simplicity; computational speed (learning time is linear in the

number of examples); potential for on-line use; and inherent robustness to noise and irrelevant attributes [17]. Noteworthy accuracy has been achieved for text classification with Naive Bayes on many natural domains, and a large body of research has shown Naive Bayes to be competitive with more sophisticated rule-induction and decision-tree algorithms; see, for example, [1].

In text classification the task is to assign a document to a particular class, typically using the so-called “bag of words” approach, which profiles document classes based simply on word frequencies [29]. Deciding whether a newspaper article is about sports, health or politics, based on the counts of words in the article, seems analogous to the task of deciding whether or not a stream of commands issued at a computer terminal belongs to a particular authorized user (who is likely to use various commands idiosyncratically). Despite its simplicity and success in text classification, hitherto the Naive Bayes approach has not been applied to user profiling with command-line data and masquerader detection.

The success of Naive Bayes has often struck researchers as surprising, given the unrealistic assumption of attribute independence which underlies the Naive Bayes approach. However, [4] demonstrates that Naive Bayes can be optimal even when this assumption is violated.

17.4.2 Description of Naive Bayes

In the present context, the classifier works as follows. The model assumes that the user is a multinomial machine,¹ generating sequences of commands, one command at a time, where each command has a fixed probability that is independent of the commands preceding it (this independence assumption is the “naive” part of Naive Bayes). The probability for each command c for a given user u is based on the frequency with which that command was observed in the training data, and is given by:

$$\theta_{c,u} = \frac{\text{Training Count}_{c,u} + \alpha}{\text{Training Data Length} + (\alpha \times A)}$$

where $\text{Training Count}_{c,u}$ is the number of times command c was seen in the training data of user u , α is a pseudocount, $\text{Training Data Length}$ is the total number of commands issued during the training phase (the same for all users in this dataset) and A is the number of distinct commands (i.e., the alphabet size) in the data. The pseudocount can be any real number larger than zero (0.01 in this study), and is added to ensure that there are no zero counts; the lower the pseudocount, the more sensitive the detector is to previously unseen commands. The pseudocount term in the denominator compensates for the addition of a pseudocount in the numerator.

Assuming independence, the probability of a test sequence of L commands in which command c appears N_c times, drawn with replacement from an alphabet of

¹The multinomial distribution is an extension of the binomial distribution. A binomial distribution has two discrete outcomes, as in a coin toss; the probability of either outcome does not change, irrespective of how many times the coin is tossed, and each outcome is independent of previous outcomes. The multinomial distribution generalizes this concept to situations in which there are more than two discrete outcomes [14].

A possible distinct commands distributed according to the probabilities of occurrence observed in the training data of user u is:

$$L! \prod_{c=1}^A \frac{(\theta_{c,u})^{N_c}}{N_c!}$$

This probability is the product of two parts. One part contains the factorials, the other the θ 's. The former is a function of the sequence alone; the latter is a function of the user's habits and the counts in the sequence. The first part is largest for high-entropy sequences. This facet of the multinomial model results in an intrinsically low probability for certain sequences, for example a sequence of identical commands, irrespective of the user's habits. For classification, it is desirable for the probabilities used to be a function of the user's habits alone. For this reason, the factorial component of the formula was neglected and sequence probabilities were calculated simply as $\prod_{c=1}^A (\theta_{c,u})^{N_c}$. Thus, the *conditional probability of a test sequence of the five commands "a a b b b" given the user u1*, is:

$$P(aabbb \mid u1) = \theta_{u1,a} * \theta_{u1,a} * \theta_{u1,b} * \theta_{u1,b} * \theta_{u1,b}$$

or $(\theta_{u1,a})^2 * (\theta_{u1,b})^3$ where $\theta_{u1,a}$ is the probability that User1 typed the command a .

In order to detect masqueraders based on unlabeled test sequences, it is necessary to know the *conditional probability of the user given the sequence*. The probability of the user given the sequence can be obtained from the probability of the sequence given the user by putting a uniform prior on the users and invoking Bayes' rule for conditional probabilities. Assuming that all users are equally likely, by Bayes' rule one obtains:

$$P(\text{User} \mid \text{Sequence}) = \frac{P(\text{Sequence} \mid \text{User})P(\text{User})}{P(\text{Sequence})}.$$

For a given test sequence $P(\text{Sequence})$ is fixed and $P(\text{User})$ is defined to be the same for all users, i.e., the probability of the user given the sequence is the same as the probability of the sequence given the user, up to a constant which can be ignored.

For each User X, a model of Not X can also be built using training data from all other users. The probability of the test sequence having been generated by Not X can then be assessed in the same way as the probability of its having been generated by User X. The larger the ratio of the probability of originating with X to the probability of originating with Not X, the greater the evidence in favor of assigning the test sequence to X. The exact cutoff for classification as X is the ratio of probabilities below which the likelihood that the sequence was generated by X is deemed too low. This cutoff can be determined by a cross-validation experiment, calculating the probability ratios for sequences which are known to have been generated by self. The range of values covered by these legitimate sequences is examined, and a cutoff is chosen.

Further details regarding Naive Bayes can be found in [22] and [30].

17.5 Data

The data used in the present study are those employed in the Schonlau et al. masquerade-detection study; they are downloadable from <http://www.schonlau.net>, or available from the authors.

As described by Schonlau et al., the data consist of 15,000 commands, from each of 70 different users, recorded over a period of several months. Some users generated their 15,000 commands in a few days; others took a few months, although details of which users took how long are not provided with the dataset. The commands were extracted from the UNIX `acct` auditing mechanism. Examples of commands are: `sed`, `eqn`, `troff`, `dpost`, `echo`, `sh`, `cat`, `netstat`, `tbl`, `sed`, `eqn`, `sh`. Some commands not explicitly typed by a user were included as well (e.g., those generated by shell files or scripts); also included were names of executable, user-defined programs (e.g., a program that a user might have named *foo*).

Fifty of the seventy users were chosen as subjects. For these users, the first 5000 commands are contiguous and may be used for training a detector. The last 10,000 commands are broken into blocks of 100 commands each (the number 100 was chosen for simplicity's sake, according to Schonlau (pp. 60)). These 100 blocks, to be used for testing, were injected with "masquerader data" drawn from 100-command blocks of the remaining 20 users; that is, data blocks from the other 20 users were employed as if these 20 users were masqueraders. The injection methodology was as follows: with a probability of .01, a user block was replaced with a masquerader block. If such a replacement was made for a given block, then the next user block would be replaced with a masquerader block (from the same masquerader) with a probability of .8. Each block is either pure user or pure masquerader - there are no mixed blocks. Because the injected replacements were done probabilistically, each of the fifty injected users contained different numbers of masquerader blocks; some users had no injections, whereas other users had as many as 24. The data thus configured will be referred to as the SEA (Schonlau Et Al.) configuration.

17.6 Improvements on previous results

17.6.1 Methodology

A separate classifier was built for each of the fifty users, individually called User X when not referring to any user in particular. During the training phase, the classifier built a profile of self and a profile of non-self (i.e., calculated the parameters for a multinomial model of command generation based on the command frequencies observed in the training data), using the first 5000 commands of User X's data for the former and the 49 x 5000 training data commands of the other 49 users for the latter profile.

During the testing phase, the classifier was presented with 10,000 commands split into either 100 unlabeled blocks of 100 contiguous commands, or 1000 unlabeled

beled blocks of 10 contiguous commands.² This testing data was predominantly self data, but had been randomly injected by Schonlau et al. with between 0 and 2400 commands (in contiguous blocks of a maximum length of 100) generated by another user, as a proxy for a masquerader. Each test block was classified by the detector as self or non-self, according to whether the ratio of the log probabilities assigned to the block under the self and non-self models was greater than a threshold (non-self) or less than or equal to that threshold (self).

An initial experiment was conducted during which the profiles of self and non-self were not updated to reflect the information contained in the test blocks. A second experiment introduced a very straightforward method of updating. The command frequencies for each block of data identified as self were passed back to the detector to update the profile of self, whilst information from each block of data classified as non-self was used to update the model of non-self. No attempt was made to update the threshold, nor was information from blocks identified as masquerader material fed back into the model of non-self.

The threshold value for self, against which a test block could be compared to decide whether it should be classified as self or non-self, was established on the basis of five-fold cross-validation, as detailed here. The training data for User X were split into five different sets of 4000 and 1000 commands, for training and testing respectively. Labeling the blocks of 1000 making up the 5000 as A, B, C, D, E (in chronological order), the five sets of training and testing data were composed in the following way: ABCD and E; ABCE and D; ABDE and C; ACDE and B; BCDE and A. For each user and each set of 4000 and 1000 commands, a detector was trained on the 4000 commands of User X and 49 x 5000 commands of Not X information (i.e., the aggregated training data of the other 49 users). The remaining 1000 commands of User X data were segmented into 10 non-overlapping blocks of 100 contiguous commands or 100 non-overlapping blocks of 10 contiguous commands. Each block was presented to the detector, and was assigned a log-probability under both the self and non-self models. The five repetitions thus generated 50 or 500 (5 x 10 or 5 x 100) log-probability ratios for each user, one for each block of self data tested. The ratio of the magnitudes of these log-probabilities was obtained and noted for each test block and each set of 4000 and 1000 commands. The reader should bear in mind that the logarithms of the probabilities were actually used for calculations; thus, perhaps counter-intuitively, the larger the ratio of self to non-self, the greater the likelihood that the block was generated by non-self. To minimize false alarms, it is sensible to take the value corresponding to the least likely block of self data, i.e., the maximum ratio value obtained over the five repetitions, as the threshold for a given user. This was the strategy used for classifying blocks of 100 commands. However, the variance in probabilities for blocks of 10 commands is much greater than for blocks of 100, and taking the maximum therefore leads to a high threshold, thus increasing the

²The split into blocks of 100 commands each has no intrinsic significance for these experiments. It was made so that the experimental results would be directly comparable with Schonlau's previous results. The alternative block size of 10 was equally arbitrary; it was chosen to illustrate what happens at smaller, more realistic block sizes, since one wishes to detect masqueraders in as few commands as possible.

chances of accepting a block of non-self as self. For blocks of 10 commands, the threshold for each user was therefore taken to be the mean of the five maximum values obtained over the five repetitions.

Preliminary experiments indicated that using a different, individualized threshold for each user often yields worse results for individual users, as well as over the dataset as a whole. This can happen when a user's behavior is more variable in the training phase than in the testing phase. Under these circumstances, cross-validation indicates the need for a high threshold to minimize false alarms. However, the regularity of the user's behavior in the true testing phase (as opposed to the cross-validation testing phase) means that no advantage is gained by the high threshold in terms of false alarms; what accrues is only the disadvantage in terms of missed intrusions due to non-self blocks slipping under the high-threshold bar. Therefore, it was decided to employ a generic threshold for all users. Although there are many ways in which a generic threshold can be obtained, due to the early and exploratory nature of our investigation it was decided to employ the simple approach of taking the mean of the individual thresholds derived from the cross-validation process described above. It is worth noting that the 100 likelihood ratios obtained for each user from cross validation do not conform to any standard distribution, so that establishing a threshold with more advanced statistical methods would have involved a selection of smoothing methods which would have been complex and not necessarily any less ad hoc than the straightforward use of the mean.

17.6.2 Results

Summary results

The overall performance of the Naive Bayes classifier on sequences of length 10 as well as sequences of length 100, with and without updating, is summarized in Table 17.1. The percentage of hits, misses and false alarms per user is detailed in Tables 17.8, 17.9, 17.12 and 17.13, located in Section 17.10. Section 17.10 also contains Tables 17.10 and 17.11, matrices which reveal the exact locations in the data stream of hits, misses and false alarms for each user (note that constructing such matrices was only possible for the experiments in which the data were partitioned into 100 blocks of 100, since it is impossible to print a readable matrix with 1000 columns on a single page).

Sequence length	100 commands		10 commands	
Updating	OFF	ON	OFF	ON
Hits %	66.2	61.5	55.2	47.1
Misses %	33.8	38.5	44.8	52.9
False Alarms %	4.63	1.3	4.7	1.6

Table 17.1: Summary results of Naive Bayes detector on sequence lengths 10 and 100.

Comparison with prior work

Table 17.2 shows the results obtained by Schonlau et al. with a variety of detectors for sequences of 100 commands (all of their detectors used updating).

Method	Hits %	Missed Alarms %	False Alarms %
Bayes 1-Step Markov	69.3	30.7	6.7
Hybrid Multi-Step Markov	49.3	50.7	3.2
Uniqueness	39.4	60.6	1.4
IPAM	41.1	58.9	2.7
Sequence Match	36.8	63.2	3.7
Compression	34.2	65.8	5.0

Table 17.2: Summary results of various Schonlau et al. detectors on sequence length 100.

In assessing the results of a masquerade detector one is concerned with the trade-off between correct detections (hits, or true positives) and false detections (false alarms, or false positives). These are often depicted on a receiver operating characteristic curve (called an ROC curve) where the percentages of hits and false alarms are shown on the y-axis and the x-axis, respectively.³ ROC curves for the Naive Bayes classifier (with updating) on sequences of length 10 and 100 with the SEA data are presented in Figure 17.1. Superimposed upon the curves are the points showing the best results reported by Schonlau et al. [34] as well as those of the present work.

The ROC curves show the results for a Naive Bayes classifier, with updating, applied to the SEA data configuration as the decision threshold was stepped through its range. The solid curve applies to the case in which the data are segmented into blocks of 100 contiguous commands; the dotted curve applies to blocks of just 10 contiguous commands. Lenient decision criteria allow a higher hit rate, but also a higher false-alarm rate; more stringent criteria tend to reduce both rates. Each point on an ROC curve indicates a particular trade-off between hits and false alarms. Points nearer to the upper left corner of the graph are the most desirable, as they indicate high hit rates and correspondingly low false-alarm rates. Recall that the unit of classification employed by both Schonlau et al. and the present study is a block of 100 contiguous commands. For the SEA data configuration as employed by Schonlau et al., the random injection methodology of the latter researchers led to between 76 and 100 blocks of unlabeled self data per user, for a total of 4769 possible false alarms, and between 0 and 24 blocks of masquerader data per user, for a total of 231 possible missed alarms. This study documents an additional experiment in which these same data were segmented into blocks of just 10 consecutive user commands. In this regime, there were between 760 and 1000 blocks of unlabeled self data per user, for a total of 47690 possible false alarms,

³For a thorough exposition of ROC curves, see [39].

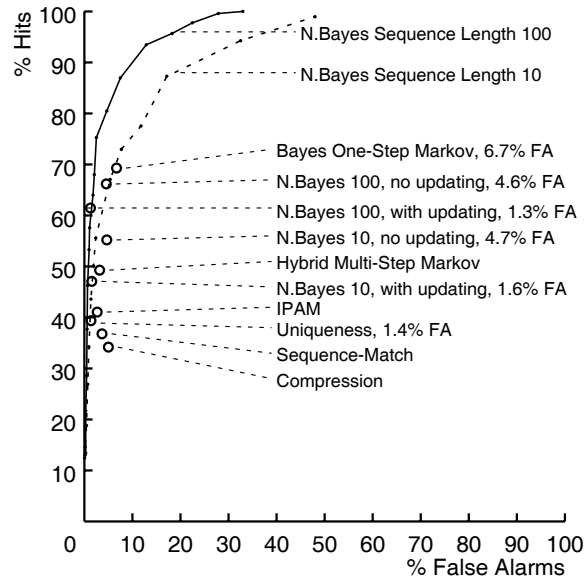


Figure 17.1: Receiver operating characteristic (ROC) curves for Naive Bayes classifier, with updating, on sequences of length 10 and 100. Superimposed are best results achieved by the various methods described in the text. Solid curve shows sequence length 100; dotted curve shows sequence length 10.

and between 0 and 240 blocks of masquerader data per user, for a total of 2310 possible missed alarms.

The goodness of a detector may be judged in terms of its ability to limit either false alarms or missing alarms. In different situations it may be appropriate to attach different weights to missed alarms and false alarms. Therefore, the overall “goodness” of each of several detection methods can be ranked by applying a cost function of the form:

$$Cost = \alpha(Misses) + \beta(FalseAlarms)$$

The cost of a false alarm in terms of misses will vary from one installation to another, so there is no obvious way to set the relative values of α and β to give an optimal cost. A first attempt at ranking the methods therefore uses a cost function in which α and β are both equal to one:

$$Cost = Misses + FalseAlarms$$

Under these circumstances, or if a missing alarm is deemed to carry more weight than a false alarm, the best-performing method for Schonlau et al. was Bayes 1-Step Markov, with 30.7 % missing alarms, and corresponding 6.7% false alarms. The performance of the Naive Bayes algorithm on the same sequence length of 100 commands compares favorably with this result, limiting missing

Method	Hits	Misses	FA	Cost
N. Bayes (Updating)	61.5	38.5	1.3	46.3
N. Bayes (no Upd.)	66.2	33.8	4.6	61.4
Uniqueness	39.4	60.6	1.4	69.0
Hybrid Markov	49.3	50.7	3.2	69.9
Bayes 1-Step Markov	69.3	30.7	6.7	70.9
IPAM	41.1	58.9	2.7	75.1
Sequence Matching	36.8	63.2	3.7	85.4
Compression	34.2	65.8	5.0	95.8

Table 17.3: Ranking of classification methods, using SEA data configuration and $\text{Cost} = \text{Misses} + 6 * (\text{False Alarms})$ as a ranking function (based on Schonlau et al. assessment of Uniqueness as their best method - see text for details).

alarms to 33.8% and achieving a better false alarm rate of 4.6%, without the need for updating.

In practice, it is often deemed more important to limit false alarms than to limit missing alarms. Thus, the Schonlau et al. experiments targeted a false alarm rate of 1%. By this standard, the top-performing algorithm reported by the Schonlau team was Uniqueness, which attained a false-alarm rate of 1.4% whilst achieving a hit rate of 39.4%. For Uniqueness to emerge as the best of the methods described by Schonlau et al., it is necessary to set the cost of a false alarm to 6 times that of a miss, i.e., to use the cost function

$$\text{Cost} = (\text{Misses}) + 6(\text{False Alarms})$$

To effect a proper comparison with the results of Schonlau et al. requires ranking the classification outcomes on the same basis as Schonlau et al. did. Hence, Table 17.3 presents the classification methods (on blocks of 100 commands) ranked according to the latter cost function: false alarms are six times as costly as misses. When false alarms and misses are weighted in this way, Naive Bayes with updating switched on, using a threshold extracted from the training data by cross validation, outperforms Uniqueness by a wide margin, achieving a 61.5% hit (detection) rate versus 39.4% for Uniqueness, an improvement of 56%, accompanied by a 7.1% improvement in false alarms, from 1.4% to 1.3%. On the basis of cost, the same Naive Bayes classifier realized a reduction of 32.9%, from 69.0 to 46.3.

The range of coefficients for the false-alarm term for which the whole ranking will be the same as that observed with a coefficient of 6, is 5.71 to 7.05. That is, if the β term in the expression above varies from 5.71 to 7.05, the rank ordering of the methods shown in Table 17.3 will not change.

Naive Bayes with updating is superior both to Naive Bayes without updating and to Bayes 1-Step Markov at any weight for false alarms greater than or equal to 1.45 times that of a miss. In other words, given the expression $x * \alpha = \beta$, Naive Bayes with updating will remain at the top of the ranking as long as $x \geq 1.45$ (although the rankings of the rest of the methods may change).

In the real world, having to wait for 100 commands before evaluating the legitimacy of a user is clearly undesirable. When the length of the sequence of commands analyzed by the detector is reduced to just 10 commands, this waiting period decreases by a factor of 10. As can be seen from Table 17.1, even with 10 times less information, Naive Bayes limits missed alarms more successfully than any of the algorithms employed by Schonlau et al. with the exception of Bayes 1-Step Markov. Not only does Bayes 1-Step Markov require 10 times as much information to achieve its 14% higher hit rate, it does so at a false alarm rate that is 2% higher than that of un-updated Naive Bayes working on sequences of size 10. In the case that false alarms are judged to have more weight than missing alarms, Naive Bayes results with sequences of just 10 commands perform very favorably again, limiting false alarms to 1.6% with a hit rate of 47.1%, a 19.5% improvement over the Uniqueness method, for only a 0.2% increase in false alarms.

Missed intrusions: what went wrong?

Table 17.10 (see appendix, Section 17.10) provides a detailed chart that breaks down the classification decisions made by Naive Bayes (without updating) on the SEA dataset, with sequences of size one hundred. Each row of the chart shows both the overall hits, misses and false alarms for a given user, and the position in the data stream of each misclassification made by the classifier for that user. The purpose of such a chart is to allow the experimenter to identify quickly the data upon which the classifier failed to do well, so that an investigation into what went wrong may be conducted. Space considerations prohibit an exhaustive analysis here, but a few of the most interesting features of the chart will be discussed.

Looking at the rightmost columns corresponding to hits and misses, one sees a large number of blank entries. Where there is no entry for either hits or misses, no masquerader data were injected by Schonlau et al. for that user. The reader is reminded that the injection methodology employed by Schonlau et al. resulted in different numbers (between 0 and 24) and types of masquerader blocks being injected into different users' data, and is referred to Section 17.5 for more details. From the columns in the chart, one learns that the worst affected users in terms of hits relative to misses were 3, 9, 12, 15 and 43. Careful examination of the graphical representation of results provided by Schonlau et al. reveals that these users suffer a similar number of misses irrespective of the algorithm employed. Three explanations for this are provided below: one for users 3, 15 and 43; one for user 9; and one for user 12.

Examining the probabilities associated with self and non-self for users 3, 15 and 43 one discovers that on the whole the masquerader blocks are distinct from the user blocks in terms of likelihood. Discrimination fails, however, because the threshold established by cross-validation is too low. Looking at the training data for these users one finds that each has a couple of rather unusual blocks. These blocks are assigned a low probability by the detector during training. In choosing a threshold, it was sought to minimize false alarms, therefore picking the lowest probability assigned to each user during the training period. The uniform threshold was then obtained by taking the mean of these minima. Thus, the presence of

a couple of unusual training blocks for a handful of users results in a uniform threshold which is lower than optimal, allowing masqueraders to pass undetected. An alternative method of choosing the threshold should improve the hit rates for these users, although it is likely to have a negative impact on the more critical false alarm rate.

Turn now to the issue of missed intrusions for User 9, the most targeted user, with 24 masquerade events. No detector identified any more than 14 of these blocks. Two masqueraders are involved here (we only know that the two *may* be different, not that they actually are). Without updating, Naive Bayes accepts 11 of the first masquerader's 22 blocks as self, and both of the second masquerader's blocks. An examination of command frequencies for the first masquerader shows that this masquerader has a few different modes of behavior. Nine of the eleven blocks from this attacker, which were erroneously accepted as self, correspond to one of the distinct modes; the remaining 2 accepted blocks correspond to another. Both of these modes of behavior seem to be accepted by the Naive Bayes detector because of the overwhelming dominance of commands `cpp`, `rm` and `ls`, all three of which fall in the top 25 commands by frequency for User 9. The same blocks of masquerader data also foil the uniqueness and compression algorithms applied by Schonlau et al., but are detected by mechanisms which make use of sequence (e.g., n-grams or tuples) as well as frequency information (as is the case for both of the Markov model detectors examined by Schonlau et al.).

Exploiting sequence information would be to no avail in the case of the missed intrusions suffered by User 12, however, due to the shared-tuple phenomenon. User 12 is not one of the top users in terms of shared tuple use (see Section 17.7.4 for details); but even so, the fact that none of the six masquerader blocks gives rise to an alarm is attributable to the presence of sequences of commands popular within the community as a whole in both the victim and the masquerader. Specifically, a certain 33-tuple is employed 8 times by User 12 and occurs 4 times in the masquerader data, affecting 3 of the six blocks. Another tuple, of length 10, crops up 33 times in User 12's training data and 9 times in the masquerader data, affecting 5 of the six blocks. The sequence of length 33 is common to the training data of 10 of the non-masquerader users in this community, whilst the sequence of 10 commands referred to is employed by 27 users during training. This case is a reminder of the dangers posed by the presence of shared sequences of commands in a user environment, such as those commands generated by system-level scripts that were installed by a system administrator.

Focusing for a moment on successful detections, one sees from the chart (appendix, Table 17.10) that all the masqueraders embedded in the data of users 10, 24, and 26 were detected by Naive Bayes. The performance of the detectors tested by Schonlau et al. was not so consistently robust with respect to these masqueraders, with the exception of the attacks against User 10, which are perfectly detected by all their algorithms except compression. In the following paragraphs it is explored why this is so.

With respect to the masquerader embedded in the data of User 10, the strong performance of every algorithm except compression is hardly surprising - the 13 blocks of masquerader data consist entirely of one long string of 1300 popper com-

mands. A compression-based system using a two-sided test (looking for blocks which are too easy to compress as well as those which are too hard) would probably perform just as well. This 5.6% of the total potential hits is really a freebie. It is reasonable to suppose that no real masquerader would issue 1300 identical commands in a row.

The situation with regard to the 21 blocks of data drawn from a single masquerader and injected into the data of User 24 is similar to that described above for the User 10 masquerader. In this case, the 2100 commands are almost exclusively long, single-command strings of `netscape` or `popper` or the trigram `netscape, popper, popper`, with an occasional smattering of `movemail`, `sendmail` and a few other commands. Since User 24 does not use any of `netscape`, `popper` or `movemail`, this masquerader is trivially easy to detect.

Two masqueraders make 10 and 3 attempts respectively against User 26. These are fairly easy to detect because the masqueraders and the user favor different commands; for example, of the 10 commands employed most frequently by the first masquerader, four are never used by User 26 in the training phase and another four are little used (fewer than 20 times) during training.

Table 17.11 (see appendix, Section 17.10) provides a detailed breakdown of the classification decisions made by Naive Bayes, with updating, on the SEA dataset, with sequences of size one hundred. With respect to the effects of updating on the hit and miss rates, the most striking feature is perhaps the loss of performance with respect to identification of non-self for User 42. Updating results in a drastic drop from 18 to only 6 hits for this user. The injection methodology employed by Schonlau et al. resulted in 20 blocks of masquerader material (drawn from a single masquerader) for this user, so the response of the detector to this user's data alone can cause the hit rate to fluctuate by up to almost 9%. Of the 20 attacks against this user, 18 are rejected by the detector when updating is switched off, but only 6 are detected when updating is turned on. The particular masquerader involved makes extensive use of bi- and trigrams made up of the command `sendmail`. This command occurs frequently for many of the users, but only infrequently in user 42's training data, so the masquerader is classified as non-self by the simple detector. However, it just so happens that the early part of the testing phase sees a sudden increase in User 42's employment of this command. When the profile built on the training data is updated to take account of this, the masquerader blocks cease to be more typical of the background model than of User 42, and are thus accepted as self. A situation like this sharpens the inappropriateness of using normal user data as masquerade data, as was done by Schonlau et al. One alternative might be to use synthetic data instead.

False-alarm analysis

Most of the observed false alarms are due to concept drift, sometimes called non-stationarity. Concept drift refers to the fact that user behavior may not be perfectly static; as time passes, new behaviors may emerge, making a user's behavior at one point in time, as represented by his profile, appear different from his behavior at a later point in time. If a user's behavior changes after the building of the profile,

the classifier may not recognize data generated by the authentic user; hence, false alarms will be raised.

Table 17.10 (see appendix, Section 17.10) provides a detailed breakdown of the classification decisions made by Naive Bayes, without updating, on the SEA dataset, with sequences of size one hundred. Looking at the column of false alarms, one sees that the majority of alarms are due to just a handful of users, namely 10, 12, 13, 16 and 20. These five users contribute 142 of the 221 false alarms, i.e., 64%. Comparing the positions of the exclamation points (indicating false alarms) in the row for any of these users to the appearance of the scattergram for that user (e.g., User 10 in Figure 17.2), reveals the problem: substantial patches of behavior are not represented in the training data. This raises the question of what happens when updating is introduced; this is addressed below, with reference to Table 17.11.

Looking at the column giving the number of false alarms per user for the Naive Bayes detector with updating switched on (Table 17.11, appendix, Section 17.10), one sees that false alarms drop from 25 to 1 for User 12, from 36 to 4 for User 13, from 32 to 4 for User 16 and from 31 to 3 for User 20. This is a 90% reduction in the false alarms raised for these users!

Decreasing the size of the sequences used for classification from 100 to 10 has only a small negative impact on the false alarm rate. With updating switched on, the false alarm rate for Naive Bayes classifying sequences of just 10 commands is only 0.3% worse than can be obtained with sequences of 100 commands. A large proportion of this 0.3% increase is due to increases in the false alarms due to users 18, 39 and 41. However, for each of these users, the individual false alarm rate remains well below 1%. The real problem is caused by intractable concept drift in the data of users 10 and 36. Concept drift is dramatically illustrated in Figure 17.2, showing a scattergram of position in the data stream vs. command id for User 10. This was one of the four users to whom 60% of the observed false alarms can be attributed. The drastic changes in command-line behavior illustrated on the scattergram coincide with the blocks which give rise to false alarms.

A combination of exploiting information about non-self (training the classifier on both self and non-self data) and updating the profiles is quite effective in tackling false alarms. However, the problem persists for User 10. This is partly due to the fact that the new behavior tends to cluster in just such a way that when the data are divided into blocks of 10 or 100 commands, there are no blocks in which the amount of data representing the new behavior is both small enough not to set off an alarm and large enough to bring about a sufficient change in the profile such that future instances of this behavior are accepted as self. The other problem with User 10 is that the new behavior exhibited by this user is actually common in the community as a whole, so that a classifier trained on self and non-self will preferentially assign those blocks to the non-self model. This is another illustration of the dangers posed by the presence of shared tuples (see Section 17.7.4 for details about shared tuples).

Users 10 and 36 contribute nearly one third of the total false alarms, with or without updating, and at either sequence length, putting a lower limit of 0.5% on the false alarm rate. It is interesting to note that the same users contributed 11 and 22 percent of the total false alarms under the Bayes 1-Step Markov and

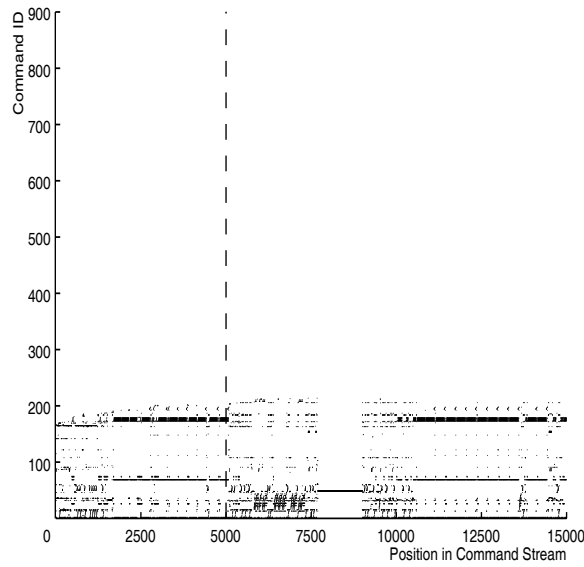


Figure 17.2: Concept drift illustrated by User 10: scattergram representation of commands. Vertical dashed line indicates division between training data (left of line) and testing data. Note regions of test data bearing no resemblance to any portion of training data.

Compression regimes tested by Schonlau et al.; by themselves, these users gave rise to false alarm rates of 0.73% and 1.09% under those two methods respectively. Unfortunately, working with truncated command lines alone (i.e., command lines whose arguments have been stripped away), there does not appear to be any way to spot these troublesome users in advance; the training data do not contain any warning of the behavioral discontinuities which will appear in the test data. Perhaps such an indicator could be found in another source of information about the users, for example degree of programming experience or type of work.

With respect to users 10 and 36, all the methods tested on the SEA data configuration have done very well in identifying masquerade attacks. However, due to the random source injection protocol employed by Schonlau et al. in creating the test set, it is not possible to know whether this is due to some facet of the behavior of the users in question, or simply a fortuitous user-masquerader (self/non-self) pairing, where the randomly selected user just happens to have behavior that is distinct in every way from that of the victim. It is difficult to draw sensible generalizations of the kind necessary to improve upon results when every victim confronts a different masquerader. To get a better handle on questions like this, the present chapter includes experiments with an alternative data configuration, termed 1v49, which is described in Section 17.7.

17.7 Insight into masquerader success

17.7.1 Motivation

A major shortcoming in the methodology followed by Schonlau et al. is that the design of the test set precludes sensible error analysis. Different masqueraders were injected into different users, and not all users were injected with masqueraders. In the real world, it is likely that only a few users will be subject to masquerader intrusions, possibly by a different masquerader in each case. Schonlau et al. sought to reflect these realities in the design of their test set. However, in a test setting, failure to run a consistent set of masqueraders against all users makes it difficult to draw constructive inferences from the hit and miss rates. When an algorithm fails to identify a masquerader block that occurs only in the data of a single victim, one does not know whether the failure is due to characteristics of the victim, the masquerader, or the algorithm. Perhaps data drawn from masquerader Y is particularly hard to detect when embedded in the data of User Z, but rather easily flagged as anomalous in the context of User W's normal behavior. If, in addition, the number of blocks of masquerader data varies from user to user, with particularly heavy concentrations in just a few of the users' streams, there is the potential for large fluctuations in the hit and miss rates simply due to experimental design. Not only was the number of masquerade events different for different users in the Schonlau et al. data, but in the majority of cases, where there were multiple masquerade events they were largely drawn from the same masquerader (no user was injected with data from more than three different masqueraders). For example: 22/24 of User 9's, 21/21 of User 24's, and 20/20 of User 42's masquerader blocks were drawn from a single masquerader. Note that these masqueraders constitute 27.3% (63/231) of the total injected intrusions. For each of the methods reported in the Schonlau et al. paper, a substantial proportion (15%, 16%, 21% and 14% respectively for Bayes 1-step Markov, Uniqueness, Hybrid Multi-step Markov and Compression) of the hits achieved were achieved by repeatedly identifying blocks taken from the same masquerader. Choosing the masquerader-user pairs differently might have had a significant impact on the success profiles reported.

17.7.2 Methodology

To address the methodological defect discussed above, the data were reconfigured for the current study to yield a test set that was consistent in the number and origin of the masquerade events encountered by each detector, i.e., if the detector that was trained on User X encountered masquerade events from 5 different masqueraders, the detector trained on User Y would encounter the same 5 events. In the new configuration, the training data remained the same as in the SEA configuration, i.e., the first 5,000 commands of a given user. However, the SEA testing data was altered. In the new configuration, the data for testing non-self recognition were 245,000 commands of testing data for each user, made up of the first 5,000 commands of the other 49 users (as if all 49 of the other users were masqueraders). The new data configuration is therefore referred to as the 1v49 (1 versus 49) configuration. It resulted in 2450 blocks of 100 contiguous masquerader commands (245,000/100)

for each user, compared with between 0 and 24 for Schonlau et al. The test of self-recognition remained the same as in the SEA configuration, with between 76 and 100 blocks of self data culled from the second 10,000 commands being presented to the detector.

In the new data configuration, for each User X the training data from the other 49 users becomes testing data for non-self, that is the ability of the detector for User 1 to identify non-self is tested on 50 blocks of data from each of users 2 through 50. This means that the training data of those 49 users cannot be used to allow the detector to build a profile of non-self, as well as of self. Thus, the Naive Bayes algorithm employed in this experiment differs from that employed in the experiments on the SEA data configuration in that it trains only on self - this is referred to as One-Class Training Input, as opposed to Two-Class Training Input. The One-Class Training Input classifier only computes the probability of a test block under a model of self, i.e., the threshold value is a single probability rather than a ratio of two probabilities. The threshold for this experiment was chosen by five-fold cross-validation - the detector for User X was trained and tested on five different divisions of User X's training data into 4000 and 1000 commands, resulting in $5 \times 1000/100$, i.e., 50 probabilities, one for each test block under the model of self built based on the training data. The probability corresponding to the block of self data that was least likely to have come from the profile recorded was taken to be the threshold value. Once again, the reader is reminded that the magnitude of the logarithm of the probability was used, so the value corresponding to the least likely block would actually be the maximum of the fifty values obtained. The mean of the maxima from all the users was then taken as the generic threshold for all the users.

By way of comparison, in order to gauge whether the defects observed in the SEA data configuration were in fact causing any large swing in the results, the One-Class Training Input detector was also run on the SEA configuration. The results are reported in Table 17.4, below.

17.7.3 Results

Configuration	Hits %	Misses%	False Alarms %
1v49	62.8	37.2	4.63
SEA	61.5	38.5	4.63

Table 17.4: Summary results of one-class training input; Naive Bayes detector on SEA and 1v49 data configurations.

The test of self-recognition capacity is the same in both data configurations, hence the false alarm rate is the same. It is reassuring that the performance of the algorithm is reasonably stable over the two different data configurations. The slight improvement in performance on the 1v49 dataset is particularly encouraging since this is the more rigorous test of the ability to identify non-self. The 1v49 data configuration requires the detector for a given User X to identify 2450 separate blocks

of intrusive data, drawn from 49 different masqueraders, compared to between 0 and 24 blocks from a maximum of 3 different masqueraders for the SEA configuration. The variety of masquerader behavior which the detector must discriminate from legitimate self behavior is therefore greater with the 1v49 configuration, and the non-self recognition test is thus more rigorous.

The rankings provided in Table 17.3 suggest which of the several classifiers (detectors) might be the best performer in a given situation, but they don't reveal anything about the sources of classification error. The 1v49 experiment, described in this section, was designed explicitly to investigate the Naive Bayes classification errors, and provide insight into such questions as what makes a successful masquerader, or what makes a user an easy or a hard target.

17.7.4 Error analysis

The base results of the 1v49 experiment are: 62.8% hits, 37.2% misses, and 4.63% false alarms, making the 1v49 outcome roughly equivalent, in terms of classification accuracy, to Naive Bayes (no updating) in the SEA version of the experiment. A detailed breakdown of the results of the 1v49 experiments, on a user by user and block by block basis, is provided in Section 17.10, Tables 17.14, 17.15, 17.16 and 17.17. The error analysis draws on the information contained in Table 17.14 and Table 17.16. Representations like the one in Table 17.16 are often called confusion matrices or misclassification matrices, because they show which users were confused with which other users (or which users were misclassified as being which other users). To facilitate explanation, an excerpt of the confusion matrix is depicted in Table 17.5. The number in each cell of the table indicates the number of missed detections when one user (victim, or self) is intruded upon by another (intruder, or non-self). For example the 46 in row 9, column 5 indicates the number of times that User 5 was undetected when masquerading as User 9. The victims (self) are listed down the rows; the intruders (non-self) are listed across the columns. Note that the numbers along the diagonal, as expected, are all zeros.

The far right column of Table 17.5 shows the total missed intrusions for each victim, indicating the victim's susceptibility to attack across the population of 50 masqueraders. The bottom row shows the total number of successful intrusions for each intruder, as indicators of attacker success. With such a table, it is easy to determine the most egregious masquerader as well as the most susceptible victim. For example, *within this portion of the table*, User 5 was the most successful masquerader, with 1617 incursions that were undetected, while User 7 showed the greatest susceptibility as a victim, with 1299 missed detections.

Examination of the full confusion matrix (Table 17.16, appendix, Section 17.10), reveals several interesting observations, addressed in detail below. User 30 suffered no successful attacks as a victim, and was never successful as an attacker; User 38 was the most successful attacker, with 1649 undetected attacks (out of 2450 attempts); and User 12 was most often victimized, with 1474 erroneously-accepted attacks.

VICTIM	INTRUDER											MISSED
	1	2	3	4	5	6	7	8	9	10	...	INTRUSIONS
1	0	2	40	47	41	1	40	3	50	9	...	1296
2	0	0	3	3	4	0	4	6	0	1	...	359
3	26	0	0	15	12	1	16	1	3	9	...	549
4	35	3	2	0	44	0	36	14	48	5	...	1110
5	10	3	2	18	0	0	32	3	34	2	...	1028
6	13	0	0	1	0	0	9	0	9	5	...	212
7	36	3	4	44	50	2	0	3	49	4	...	1299
8	2	1	2	38	36	0	9	0	14	0	...	611
9	37	3	2	45	46	1	40	2	0	14	...	1162
10	9	0	2	25	0	3	14	2	26	0	...	365
...
SUCCESSFUL INTRUSIONS	1124	402	339	1405	1617	74	1379	360	1359	270	...	

Table 17.5: Confusion matrix (excerpt): simple Naive Bayes classifier without updating, 1v49 configuration. “Self” is listed down the left column; “non-self” is listed across the top. Numbers in the matrix indicate misses – the number of times the detector erroneously accepted a masquerader (non-self) as self, e.g., User 5 successfully masqueraded 46 times as User 9.

What makes a masquerader successful?

Table 17.16 (appendix, Section 17.10) shows that the most successful masqueraders were users 38, 05 and 25. These masqueraders were successful across the board; more than half the users falsely accepted 35 or more of these masqueraders’ blocks as self, leading to success rates of 67% (1649/2450), 66% (1617/2450) and 64% (1578/2450), respectively.

What could make it so easy for these masqueraders to imitate the other users? As a first step towards answering this question, scattergrams were made of the position in the sequence of 5000 training commands (which can loosely be regarded as the time axis) against the command employed at that position. The scattergrams for users 38 and 5 are shown as Figures 17.3 and 17.4.

The scattergrams for these and a number of other users had a striking feature - patterns of wavy lines which suggested the presence of repeating sequences of commands in the data. Preliminary investigations of the data by hand indicated that the data did in fact contain sequences of commands repeatedly employed by multiple users. These sequences were dubbed “shared tuples”, and a thorough investigation of the extent of their occurrence and the effect of their presence on detection accuracy was launched.

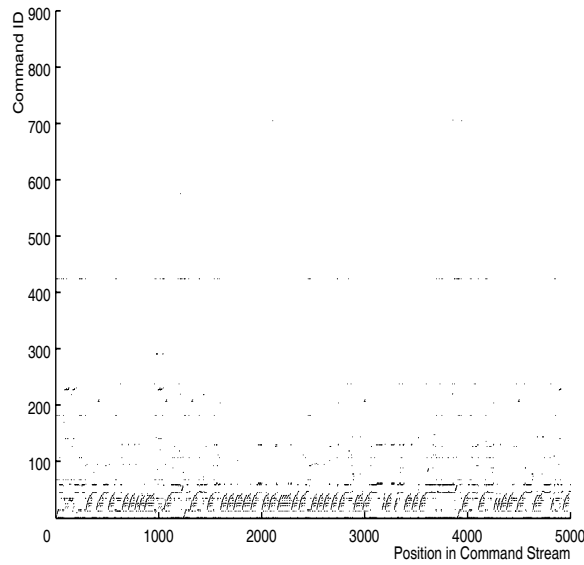


Figure 17.3: User 38: scattergram representation of training data.

Shared-tuple investigation

A sliding window facilitated identification of all the sequences of between 2 and 100 commands which were present in the training data as a whole. Sequences which did not occur in the data of at least 10 users, and sequences which only occurred as a subset of a longer sequence, were discarded. Seventy-two such non-overlapping sequences were identified. It was found that these shared sequences, or tuples, accounted for 34.75% of the training data as a whole – an enormous proportion. The longest shared tuple meeting the 10-user criterion contained 49 commands, and was employed by 18 users. Over 60% of the training data for User 38 was accounted for by such shared sequences. By way of example, the following 10-tuple occurred multiple times in the data of 27 users: `hostname, id, awk, getopt, true, true, grep, date, lp, find`.

Having compiled a list of shared tuples, the proportion of each user's training data which had been generated by employing these shared sequences was calculated; the amount of data attributable to the use of shared tuples is referred to as coverage. Table 17.6 shows the proportion of the training data for each given user, made up of tuples shared by at least 10 users; i.e., the coverage of such tuples. Correlating the coverage with the number of times a user succeeded in passing himself off as one of the other users, the linear relationship between the two had a P-value less than .0001 ($R^2 = .52$, $r_{ho} = .72$), indicating a correlation not due to chance. In other words, use of sequences of between 2 and 49 commands which are found in the data of 10 or more users explains 52% of the observed variation in the ability of the 50 users to masquerade as one another. Figure 17.5 shows the correlation between tuple coverage and the number of blocks of data that each user was able

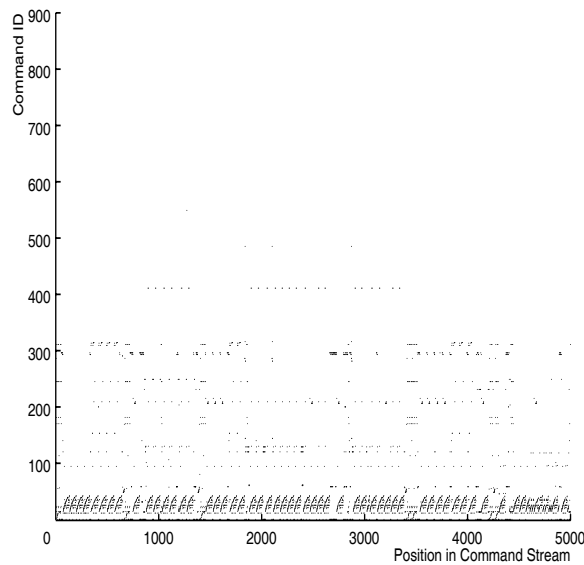


Figure 17.4: User 5: scattergram representation of training data.

to pass off as other users, i.e., masquerade success.

In terms of coverage of the training data by tuples common to 10 or more users, the top masqueraders are ranked 2nd, 12th, 14th and 4th respectively. The order of success does not exactly follow the order of coverage by shared tuples; that is, the correlation is not perfect, because the popularity of the tuples concerned (i.e., the number of users sharing the tuple) also counts. Why is it that some users with a lot of shared tuples do not do well as masqueraders? Referring to the interesting points on the correlation plot: Why isn't User 25 a more successful masquerader than User 5? Why aren't users 16 and 21 more successful as masqueraders? Why is User 31 successful?

User 38 uses more and more-popular (popularity is measured in terms of the number of users employing a tuple) shared tuples than any of the other masqueraders. The three tuples with the most coverage in this user's training data make up 31% of that data and have an average popularity of 22. The shared tuples employed by User 37 are very popular (the top tuples in terms of coverage have an average popularity of 22), but occur more sparsely than for the other top masqueraders (making up only 18% of User 37's training data). Over half of User 25's training data is made up of popular shared tuples. However, User 25's data also contains many blocks largely composed of long strings of command 48, *netscape*. The frequency with which User 25 employs the *netscape* command is anomalous with respect to the rest of the community. The average frequency with which this command is used by members of the community after removal of User 25 is 185, whereas User 25 employs the command 1266 times! This causes User 25 to enjoy markedly less success than the other top masqueraders in masquerading against users 8, 19, 24, 28, 31, 34 and 44, none of whom employ *netscape* commands to

Rank	User	Coverage %
1	16	70.5
2	38	66.6
3	21	58.1
4	25	57.9
5	41	55.1
6	14	54.1
7	15	52.2
8	22	51.4
9	1	51.2
10	19	50.4
11	40	48.7
12	48	48.6
13	5	48.5
14	37	45.4
15	9	44.2
16	17	42.4
17	4	41.9
18	39	41.7
19	33	41.1
20	13	40.1
21	28	39.7
22	31	39.5
23	7	38.4
24	45	37.8
25	43	37.4
26	47	37.3
27	2	37.2
28	50	36.7
29	44	35.8
30	49	34.2
31	12	33.9
32	11	32.1
33	24	29.5
34	34	29.2
35	20	27.1
36	23	26.6
37	26	25.0
38	3	23.0
39	42	22.3
40	6	20.2
41	29	19.8
42	27	16.7
43	10	14.4
44	35	13.3
45	8	9.3
46	18	3.9
47	46	3.5
48	32	2.1
49	36	1.3
50	30	0.0

Table 17.6: Ranked percentage of training data covered by tuples common to at least 10 users.

any significant extent (the command is not featured in a list of the top 25 commands by frequency for any of these users).

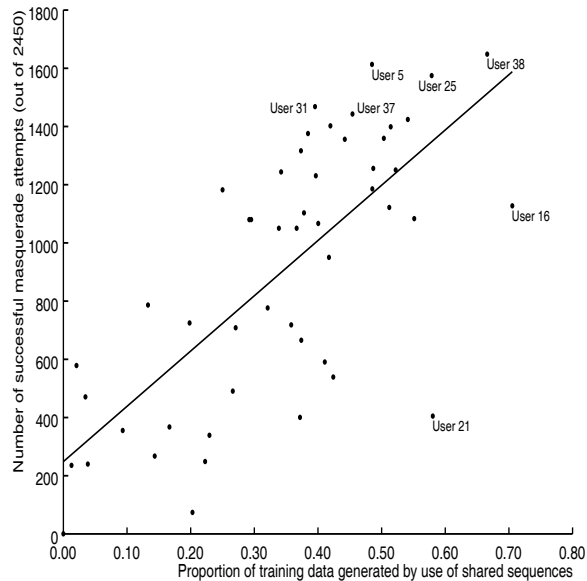


Figure 17.5: Correlation between use of shared tuples (common to 10 or more users) and masquerader success. To avoid clutter, only selected points are labeled with a user number.

User 16 has the highest coverage by shared tuples - a whopping 70.5% of this user's training data is accounted for by tuples common to 10 or more users. However, User 16 ranks 20th in terms of masquerader success. User 16's failure as a masquerader can be attributed to this user's unusually frequent use of strings of `netscape` and of the bigram `189,189 (gcc, gcc)`. User 16 employs `popper` 414 times, compared to an average of 185 over the other users, and uses `command gcc` 532 times, compared to an average of 63 over all users. Non-overlapping tuples made up only of these two commands make up 17% of User 16's training data. A further 23% of User 16's training data is accounted for by the shared 5-tuple `uname, awk, cpp, cc1, as`. This tuple occurs very frequently in the training data as a whole, but is only shared by 12 users.

A high frequency of the same 5-tuple is behind the poor performance as a masquerader of User 21, despite 58% coverage by shared tuples. In this case, the 5-tuple `uname, awk, cpp, cc1, as` accounts for an enormous 37% of the user's training data.

User 31 is as successful as users 37 and 25, despite a substantially (20%) lower coverage by shared tuples. This may be explained by the fact that although shared tuples make up only 40% of this user's training data, those shared tuples which user 31 does employ are highly popular - the three tuples which occur most frequently in User 31's data have an average popularity of 23.

In summary, a careful analysis of the data reveals that good masqueraders suc-

ceed against the Naive Bayes detector by virtue of one or more of the following attributes: heavy use of popular shared tuples; moderate use of very popular shared tuples; absence of anomalously high-frequency use of any particular command; and absence of frequent use of unpopular tuples. Shared tuples are a serious problem. The reader is reminded that the amount of training data accounted for by shared tuples ranges between 0 (for User 30) and 70.5% (for User 38), for a total of 34.8% of the training data as a whole. 52% of the variation in ability to masquerade can be explained by the amount of the user's data that is made up of shared tuples. The more shared tuples, the greater the user's chances of success as a masquerader. What is learned from the scattergram of Figure 17.5 is that if an insider wishes to thwart any system which uses command frequencies, command popularities or n-tuple frequencies, and works in an environment with many built-in, widely-used scripts, he need only invoke a few of these scripts to generate a slew of commands with which to camouflage any unusual commands resulting from his nefarious activities. Shared tuples, arising perhaps from scripts built in to the computing environment, are very effective camouflage for the insider masquerader. This observation suggests that monitoring and removal of scripts common to the environment during data collection would be an important element in improving detection accuracy. A brief investigation into the effects of "de-tupling" the data is described next in Section 17.7.4.

Experiment with de-tupled data

The 1v49 experiment revealed that the presence of shared tuples in the data adversely affects detection accuracy. Thus, it seems reasonable to examine the effect of removing the tuples from the dataset.

Complete removal of all the tuples (sequences of 2 or more commands shared by 10 or more users) leads to a drastic reduction in the amount of training data available for certain users. The worst affected is User 16, for whom just 1473 commands remain after all of the shared tuples have been stripped out. An alternative to entirely stripping out the shared tuples is to map each distinct shared tuple to a new command. When this is done, User 38 turns out to suffer the greatest reduction in training data, with only 2027 of the original 5000 commands remaining.

Clearly it is not possible to conduct an experiment with the de-tupled data which will yield results directly comparable with those of previous experiments on this dataset. Nonetheless, it seems reasonable to run a few experiments with data from which shared tuples have been removed, either by stripping them out entirely or by condensing each tuple to a single, new command, in order to get a feel for how the accuracy of detection may be improved by processing command-line data in this way.

To this end, the locations of those sequences of commands common to at least 10 users identified by the investigation described in Section 17.7.4 were established for each user. The training data were then processed in two ways: first by removing such sequences entirely; and second by replacing each such sequence with a single, new, unique command. Thus, three classes of training data were created for each user: data with shared tuples, data without shared tuples, and data with condensed

shared tuples. Within each class, the data were then further processed by truncation to just 1000 commands.

Five-fold cross-validation was performed on the three classes of training data, using sets of 800 and 200 commands, and a sequence length of just 10 commands, in order to establish thresholds. For each User X and each class of data, a Naive Bayes detector was then trained on 1000 commands of User X data, and subsequently tested on 49 x 1000 commands drawn from the other 49 users. These test data were presented to the detector in the form of blocks of 10 commands.

The experiment was intended to investigate the effect of different treatments of the shared-tuple problem on the hit rate. False alarms were not investigated. The results are presented in Table 17.7. The table shows that misses were reduced by 23.7% when the shared tuples were completely removed, going from 31.2% misses in the intact data to 23.8% missed in the de-tupled data. At the same time, the hit rate increased by 10.8%, from 68.8% in the intact data to 76.2% in the de-tupled data. This suggests that at least moderate performance improvements can be obtained simply by ensuring that tuples are not shared among users. In the context of previous work in masquerade detection, improvements of this magnitude could be regarded as quite substantial. Of course the improvements shown here could be higher if the criteria were less strict. The removed or condensed tuples were common to at least ten users; many tuples are common to fewer than ten users, and if these had also been removed, performance improvements would have been even greater.

	Left intact	Condensed	Removed
Hits %	68.8	75.0	76.2
Misses %	31.2	25.0	23.8
False Alarms %	NA	NA	NA

Table 17.7: Summary results of de-tupled data experiments.

What makes a user a hard target?

The toughest users to imitate were users 30, 47 and 32. These users suffered 0% (0/2450), 3% (74/2450) and 7.6% (185/2450) misses respectively. Investigation of these users' data reveals that the characteristics which contribute to robustness against attack are: very heavy use of particular commands, regular use of rare commands, and infrequent use of popular commands. For example, more than half of the data for User 47 are made up of a single command - `cat` - occurring a total of 2565 times compared to an average over all users of just 223. In addition, User 47 employs a number of unique or rare commands, for example, `cgiparse`, with relatively high frequency, in a substantial number of blocks. User 30 has an extremely individual command-line profile. This user employs only 5 commands, of which two are barely used at all. Three commands, `tcsh`, `rshd` and `rdistd`, make up 99.6% of this user's training data. `rdistd` alone constitutes 33% of this user's training data, and is unique to this user. The other two commands have popularities

of 5 and 15 and are employed with more than 5 times the average frequency.

What makes a user an easy victim?

The users with the highest miss rates were users 12, 50 and 15. These users suffered 60.16% (1474/2450), 58.82% (1441/2450) and 57.10% (1399/2450) misses respectively. If one models average user behavior with a vector of average command frequencies, and calculates the Euclidean distance between this average vector and the frequency vector for each user's own training data, one discovers that the profiles of users 12, 50 and 15 are the second, third and fourth closest to the average. The profile closest to the average is that of User 22. This user ranks 6th in terms of miss proneness. In other words, what these users have in common is their "averageness". The correlation between proximity to the average and vulnerability is not perfect because the particular user-masquerader pairwise interactions also contribute significantly to the final tally of misses for a particular user. For example, a glance at the confusion matrix column corresponding to User 6 reveals that this user is in general a very poor masquerader, managing to pass off only one block against User 12, the most victimized user in general. The inability of User 6 to masquerade is the flip-side of this user's relative immunity to imitation by intruders. Both can be explained by regular, well-distributed use of unpopular commands; ten of the 25 commands most frequently employed by User 6 are used by four or fewer other users, and six of these ten unpopular commands occur in more than 40 of this user's 50 training blocks. Nonetheless, User 6 passes as User 42 fifty out of fifty times, and User 42 passes as User 6 forty-three out of fifty times, because these two users happen to share the same pool of otherwise very unpopular commands, and use them with similar frequency.

17.8 Discussion

Although the results achieved by using the Naive Bayes classifier constitute a 56% improvement over the previous result with the lowest false alarm rate, these improved results remain unsuitable for fielding a masquerade-detection system. The hit rate is too low, and the false alarm rate is too high. To be effective in an operational environment, a detector needs to have a false alarm rate of not more than 1%, and probably much lower than that. Acceptable false alarm rates are determined by the number of events per unit time that could raise an alarm. If a detector examines a million events per day, which for some detection environments would be considered a low number, a 1% false alarm rate would give 10,000 false alarms per day. That many false alarms cannot be handled by the limited number of analysts on duty, so the false alarm rate would need to be considerably lower. Concomitantly, hit rates must be higher to avoid missing true masquerade events. Reaching effective performance levels will require improvements in several areas:

Data. The poor performance of every method tried so far on these data (i.e., previous studies, as well as the present study) may indicate that command-line data alone, without arguments, are not enough to profile a user. Useful additions

might be: arguments to commands, type and length of sessions, or user categories (job-type, department, etc.).

Careful data collection is important. Algorithms looking at the order of commands (e.g., Markov models) may have been adversely affected by the fact that these data are not truly sequential. In `acct`, the package used to collect the data, commands are not logged in the order in which they are typed, but rather when the application ends; moreover, commands issued in different terminal windows are interleaved. Such factors can constitute a considerable confound for Markov-based or other sequence-based detectors.

Careful preliminary examination of the data is important in avoiding things like including shared scripts which are built into the user environment (e.g., system login scripts) and which reduce the individuality of the data upon which the detector is trying to build a user profile.

It should be noted that using data blocks of 100 commands is rather unrealistic, because any serious and sensible masquerader is likely to hit and run in far fewer than 100 commands. Detection within a block of ten commands would seem more realistic. This study used a block size of 100 to allow comparison with the several Schonlau et al. results; however, a preliminary investigation using blocks of size ten has shown promising results: a 50.4% hit rate with a 2.1% false alarm rate. This compares quite favorably, all considered, with the block-100 results (61.5% hits and 1.3% false alarms).

Modeling a masquerade event by injecting one ordinary user's data into another ordinary user's data is far from ideal. Obviously, a real masquerader will not simply sit down at a victim's terminal and go about his normal business, but will seek to emulate the victim as accurately as possible. Real masqueraders have objectives and methods that may be more stealthy, and hence harder to detect. However, in the absence of any real masquerader data, the Schonlau et al. approach is a pragmatic starting point for a principled assault on the masquerader problem.

What would constitute better data? Since it is not completely clear what the effect of the Schonlau et al. data set is, because there is no data set against which to compare it, one can only speculate about how to effect improvements in the data. Some ideas are:

- gather data over equal lengths of time for all users, as opposed to a few days for some users and a few months for others;
- try to balance data across users by obtaining the same number of user/login sessions for each user;
- get time-stamps for logins and for each user command issued, so that one day can be differentiated from another;
- remove shared scripts, such as system login scripts, because these scripts inject shared tuples into the user data;
- get richer user data in terms of complete user commands complete with all the command-line arguments;

- get a job description for each user (e.g., programmer, researcher, manager, secretary, sys admin, etc.); and
- improve the features in the data by using latent features instead of, or in addition to, the raw features.

Methodology. The particular way an experiment is done can have dramatic effects on its outcome; small methodological details, accidentally overlooked, can impede other researchers from replicating or evaluating results. Experimental methods can substantially influence the validity of results. For example, the way that masquerade data are composed from segments of many users may be biased, and that bias affects outcomes. In the Schonlau et al. work this is manifested by the hit rates depending on the coincidences of which masquerade blocks were injected into which users. An additional biasing factor is the differential injection rates of masqueraders into different victims; some victims were only sparsely injected, some heavily and/or from all one masquerader, and some not at all. To gain the most in terms of understanding what works, what doesn't work, and why, a uniform injection methodology is needed.

Error analysis. Error analysis is seldom done on classification or learning results. The present study makes it clear that an examination of errors can reveal important facts about what causes the detector to make a mistake, thus providing a basis for future improvement. Errors often teach more than correct results do.

Classifier. It is interesting to ask why Naive Bayes performed so much better than its competitors. Unfortunately, the answer to this question is as yet unclear. The hit rates of Naive Bayes might be explained by Naive Bayes being good at summing weak evidence. Such a situation would ensue when no single factor is particularly indicative of a masquerade intrusion, but many small things, in combination, are. The pseudocount probably helps to explain the lower false alarm rate of Naive Bayes compared with the Markov detector; around two hundred commands occur in testing that do not occur in training, i.e., there are a lot of zeroes in the historical transition probability matrix formed on the basis of the training data. Shared tuples may explain the poor performance of Uniqueness (Schonlau et al. note that with Uniqueness an intruder using mostly common commands will slip past the detection system). However, a detailed explanation for what it is that makes Naive Bayes appropriate for these data awaits further research.

17.9 Conclusion

This chapter has shown several important aspects of studying masquerader behavior as represented in monitored data – two things in particular:

1. that the design of a study (e.g., the 1v49 configuration) can influence substantially the outcome, especially when the goal is to test the efficacy of a detection scheme; and
2. that there is much to be learned from performing even a simple error analysis.

The biggest cause of false alarms was due to changes in user behavior over time (concept drift); however, this can be mitigated to a large extent (90% in this work) by incorporating a concept updating scheme in the classifier. Sequences of commands (tuples) that are shared among users can foster successful masquerades; in fact, the more tuples that a user shares with another, the more successful a masquerade attack against that user is likely to be. This shared-tuple discovery was facilitated by a simple graphing scheme which revealed patterns that appeared similar across the data of several users. Further research is needed to improve and understand classifier performance, but even more important is the gathering of competent data that truly support the undertaking of masquerade detection.

17.10 Appendix: Details of the two-class training experiments

The information in this appendix is discussed in Sections 17.6.2 (Missed intrusions: what went wrong?), 17.6.2 (False-alarm analysis), 17.7.4 (Error analysis) and 17.7.4 (What makes a masquerader successful?).

The first several tables address the two-class training condition for the masquerade detector. Tables 17.8 and 17.9 compare the Naive Bayes classifier results, using a data sequence length of 100, with updating (compensation for concept drift) turned off and on, respectively. Tables 17.10 and 17.11 contain charts that depict the exact placement of hits, misses and false alarms for each injection across all 50 users, showing the effects of having updating turned off and on, respectively. Tables 17.12 and 17.13 compare the Naive Bayes classifier results, using a data sequence length of 10, with updating turned off and on, respectively.

The rest of the tables address the one-class training condition. Tables 17.14 and 17.15 compare the Naive Bayes classifier results, using a data sequence length of 100, without updating (compensation for concept drift), on the two data configurations used in the study: 1v49 and SEA, respectively. Table 17.16 shows the confusion matrix (or misclassification matrix) for the 1v49 study. This matrix provided the information for the error analysis, including tallies of the most successful masqueraders, the hardest targets, and the easiest victims. Table 17.17 charts the exact placement of hits, misses and false alarms for each injection across all 50 users for the one-class training condition, using the SEA data configuration, sequence size 100.

User	Hits	Misses	False Alarms
1	0/0	0/0	1/100
2	3/3	0/3	0/97
3	6/11	5/11	0/89
4	2/2	0/2	0/98
5	0/0	0/0	0/100
6	0/0	0/0	0/100
7	9/13	4/13	1/87
8	0/0	0/0	0/100
9	11/24	13/24	0/76
10	13/13	0/13	18/87
11	0/0	0/0	1/100
12	0/6	6/6	25/94
13	0/0	0/0	36/100
14	0/0	0/0	1/100
15	0/6	6/6	0/94
16	4/10	6/10	32/90
17	0/0	0/0	0/100
18	6/6	0/6	3/94
19	0/0	0/0	0/100
20	0/0	0/0	31/100
21	0/0	0/0	0/100
22	0/0	0/0	0/100
23	1/1	0/1	0/99
24	21/21	0/21	0/79
25	2/9	7/9	0/91
26	13/13	0/13	4/87
27	0/0	0/0	0/100
28	3/3	0/3	2/97
29	1/1	0/1	10/99
30	3/3	0/3	0/97
31	0/0	0/0	0/100
32	0/0	0/0	0/100
33	0/0	0/0	5/100
34	3/12	9/12	0/88
35	1/1	0/1	0/99
36	6/6	0/6	10/94
37	0/2	2/2	0/98
38	6/9	3/9	5/91
39	0/0	0/0	7/100
40	0/0	0/0	5/100
41	2/3	1/3	7/97
42	18/20	2/20	6/80
43	5/16	11/16	5/84
44	6/6	0/6	2/94
45	2/5	3/5	1/95
46	4/4	0/4	0/96
47	0/0	0/0	0/100
48	2/2	0/2	2/98
49	0/0	0/0	1/100
50	0/0	0/0	0/100
Totals:	153/231	78/231	221/4769

Table 17.8: Two-class training input; Naive Bayes algorithm without updating; SEA data configuration; sequence size 100; threshold 1.263.

User	Hits	Misses	False Alarms
1	0/0	0/0	1/100
2	3/3	0/3	0/97
3	8/11	3/11	0/89
4	2/2	0/2	0/98
5	0/0	0/0	0/100
6	0/0	0/0	1/100
7	9/13	4/13	1/87
8	0/0	0/0	0/100
9	10/24	14/24	0/76
10	13/13	0/13	14/87
11	0/0	0/0	0/100
12	0/6	6/6	1/94
13	0/0	0/0	4/100
14	0/0	0/0	0/100
15	0/6	6/6	0/94
16	3/10	7/10	4/90
17	0/0	0/0	1/100
18	6/6	0/6	3/94
19	0/0	0/0	0/100
20	0/0	0/0	3/100
21	0/0	0/0	0/100
22	0/0	0/0	0/100
23	1/1	0/1	0/99
24	21/21	0/21	0/79
25	8/9	1/9	0/91
26	13/13	0/13	4/87
27	0/0	0/0	0/100
28	3/3	0/3	0/97
29	1/1	0/1	1/99
30	3/3	0/3	0/97
31	0/0	0/0	0/100
32	0/0	0/0	0/100
33	0/0	0/0	1/100
34	0/12	12/12	0/88
35	1/1	0/1	0/99
36	6/6	0/6	9/94
37	0/2	2/2	0/98
38	7/9	2/9	5/91
39	0/0	0/0	0/100
40	0/0	0/0	5/100
41	2/3	1/3	1/97
42	6/20	14/20	0/80
43	2/16	14/16	1/84
44	6/6	0/6	1/94
45	2/5	3/5	0/95
46	4/4	0/4	0/96
47	0/0	0/0	0/100
48	2/2	0/2	0/98
49	0/0	0/0	0/100
50	0/0	0/0	0/100
Totals:	142/231	89/231	61/4769

Table 17.9: Two-class training input; Naive Bayes algorithm with updating; SEA data configuration; sequence size 100; threshold 1.227.

[illegible]

User	Hits	Misses	False Alarms
1	0/0	0/0	10/1000
2	29/30	1/30	2/970
3	51/110	59/110	1/890
4	14/20	6/20	5/980
5	0/0	0/0	0/1000
6	0/0	0/0	2/1000
7	62/130	68/130	6/870
8	0/0	0/0	32/1000
9	100/240	140/240	4/760
10	130/130	0/130	164/870
11	0/0	0/0	14/1000
12	0/60	60/60	259/940
13	0/0	0/0	281/1000
14	0/0	0/0	8/1000
15	0/60	60/60	6/940
16	22/100	78/100	313/900
17	0/0	0/0	3/1000
18	58/60	2/60	66/940
19	0/0	0/0	3/1000
20	0/0	0/0	206/1000
21	0/0	0/0	0/1000
22	0/0	0/0	4/1000
23	10/10	0/10	1/990
24	210/210	0/210	6/790
25	22/90	68/90	0/910
26	99/130	31/130	32/870
27	0/0	0/0	7/1000
28	15/30	15/30	86/970
29	7/10	3/10	107/990
30	30/30	0/30	4/970
31	0/0	0/0	2/1000
32	0/0	0/0	1/1000
33	0/0	0/0	40/1000
34	10/120	110/120	7/880
35	5/10	5/10	3/990
36	45/60	15/60	95/940
37	0/20	20/20	5/980
38	60/90	30/90	54/910
39	0/0	0/0	88/1000
40	0/0	0/0	42/1000
41	14/30	16/30	57/970
42	122/200	78/200	10/800
43	28/160	132/160	45/840
44	59/60	1/60	13/940
45	15/50	35/50	71/950
46	40/40	0/40	2/960
47	0/0	0/0	7/1000
48	18/20	2/20	42/980
49	0/0	0/0	10/1000
50	0/0	0/0	31/1000
Totals:	1275/2310	1035/2310	2257/47690

Table 17.12: Two-class training input; Naive Bayes algorithm; SEA data configuration; no updating; sequence size 10; threshold 1.352.

User	Hits	Misses	False Alarms
1	0/0	0/0	10/1000
2	30/30	0/30	1/970
3	53/110	57/110	1/890
4	12/20	8/20	4/980
5	0/0	0/0	0/1000
6	0/0	0/0	4/1000
7	27/130	103/130	6/870
8	0/0	0/0	4/1000
9	57/240	183/240	5/760
10	130/130	0/130	137/870
11	0/0	0/0	2/1000
12	0/60	60/60	8/940
13	0/0	0/0	51/1000
14	0/0	0/0	7/1000
15	0/60	60/60	3/940
16	10/100	90/100	7/900
17	0/0	0/0	4/1000
18	59/60	1/60	75/940
19	0/0	0/0	2/1000
20	0/0	0/0	31/1000
21	0/0	0/0	0/1000
22	0/0	0/0	2/1000
23	10/10	0/10	2/990
24	210/210	0/210	4/790
25	70/90	20/90	0/910
26	80/130	50/130	27/870
27	0/0	0/0	4/1000
28	3/30	27/30	9/970
29	7/10	3/10	13/990
30	30/30	0/30	3/970
31	0/0	0/0	2/1000
32	0/0	0/0	1/1000
33	0/0	0/0	18/1000
34	6/120	114/120	7/880
35	5/10	5/10	2/990
36	38/60	22/60	72/940
37	0/20	20/20	2/980
38	61/90	29/90	50/910
39	0/0	0/0	30/1000
40	0/0	0/0	32/1000
41	14/30	16/30	55/970
42	40/200	160/200	5/800
43	17/160	143/160	9/840
44	54/60	6/60	13/940
45	8/50	42/50	13/950
46	40/40	0/40	5/960
47	0/0	0/0	4/1000
48	18/20	2/20	9/980
49	0/0	0/0	10/1000
50	0/0	0/0	7/1000
Totals:	1089/2310	1221/2310	772/47690

Table 17.13: Two-class training input; Naive Bayes algorithm; SEA data configuration; with updating; sequence size 10; threshold 1.342.

User	Hits	Misses	False Alarms
1	1154/2450	1296/2450	1/100
2	2091/2450	359/2450	0/97
3	1901/2450	549/2450	0/89
4	1340/2450	1110/2450	0/98
5	1422/2450	1028/2450	0/100
6	2238/2450	212/2450	0/100
7	1151/2450	1299/2450	0/87
8	1839/2450	611/2450	0/100
9	1288/2450	1162/2450	1/76
10	2085/2450	365/2450	19/87
11	1216/2450	1234/2450	1/100
12	976/2450	1474/2450	1/94
13	1270/2450	1180/2450	32/100
14	1437/2450	1013/2450	0/100
15	1051/2450	1399/2450	0/94
16	1092/2450	1358/2450	59/90
17	1186/2450	1264/2450	0/100
18	2060/2450	390/2450	2/94
19	1426/2450	1024/2450	0/100
20	1317/2450	1133/2450	22/100
21	2219/2450	231/2450	6/100
22	1107/2450	1343/2450	0/100
23	1307/2450	1143/2450	0/99
24	1487/2450	963/2450	0/79
25	1464/2450	986/2450	0/91
26	1838/2450	612/2450	3/87
27	1283/2450	1167/2450	8/100
28	1541/2450	909/2450	0/97
29	1415/2450	1035/2450	10/99
30	2450/2450	0/2450	0/97
31	1416/2450	1034/2450	0/100
32	2265/2450	185/2450	0/100
33	2065/2450	385/2450	7/100
34	1407/2450	1043/2450	0/88
35	2159/2450	291/2450	0/99
36	2218/2450	232/2450	10/94
37	1331/2450	1119/2450	0/98
38	1292/2450	1158/2450	5/91
39	1231/2450	1219/2450	13/100
40	1389/2450	1061/2450	6/100
41	1304/2450	1146/2450	5/97
42	1399/2450	1051/2450	0/80
43	1162/2450	1288/2450	5/84
44	1369/2450	1081/2450	0/94
45	1310/2450	1140/2450	1/95
46	2208/2450	242/2450	0/96
47	2376/2450	74/2450	1/100
48	1061/2450	1389/2450	0/98
49	1340/2450	1110/2450	3/100
50	1009/2450	1441/2450	0/100
Totals:	76962/122500	45538/122500	221/4769

Table 17.14: One-class training input; Naive Bayes algorithm without updating; 1v49 data configuration; sequence size 100; threshold 6.479.

User	Hits	Misses	False Alarms
1	0/0	0/0	1/100
2	3/3	0/3	0/97
3	3/11	8/11	0/89
4	2/2	0/2	0/98
5	0/0	0/0	0/100
6	0/0	0/0	0/100
7	9/13	4/13	0/87
8	0/0	0/0	0/100
9	12/24	12/24	1/76
10	13/13	0/13	19/87
11	0/0	0/0	1/100
12	0/6	6/6	1/94
13	0/0	0/0	32/100
14	0/0	0/0	0/100
15	0/6	6/6	0/94
16	1/10	9/10	59/90
17	0/0	0/0	0/100
18	6/6	0/6	2/94
19	0/0	0/0	0/100
20	0/0	0/0	22/100
21	0/0	0/0	6/100
22	0/0	0/0	0/100
23	1/1	0/1	0/99
24	21/21	0/21	0/79
25	7/9	2/9	0/91
26	11/13	2/13	3/87
27	0/0	0/0	8/100
28	3/3	0/3	0/97
29	1/1	0/1	10/99
30	3/3	0/3	0/97
31	0/0	0/0	0/100
32	0/0	0/0	0/100
33	0/0	0/0	7/100
34	6/12	6/12	0/88
35	1/1	0/1	0/99
36	6/6	0/6	10/94
37	0/2	2/2	0/98
38	7/9	2/9	5/91
39	0/0	0/0	13/100
40	0/0	0/0	6/100
41	1/3	2/3	5/97
42	10/20	10/20	0/80
43	2/16	14/16	5/84
44	6/6	0/6	0/94
45	1/5	4/5	1/95
46	4/4	0/4	0/96
47	0/0	0/0	1/100
48	2/2	0/2	0/98
49	0/0	0/0	3/100
50	0/0	0/0	0/100
Totals:	142/231	89/231	221/4769

Table 17.15: One-class training input; Naive Bayes algorithm without updating; SEA data configuration; sequence size 100; threshold 6.479.

SELF	NON-SELF																																																		MISSES
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	
1	0	2	40	47	41	1	40	3	50	9	40	30	40	37	39	28	16	38	33	24	0	37	14	20	48	38	11	34	16	0	37	11	7	22	20	2	46	48	25	41	49	5	44	16	17	2	24	42	31	31	1296
2	0	0	3	3	4	0	4	6	0	1	0	7	4	2	4	23	21	0	1	2	46	5	1	2	2	2	5	8	4	0	21	11	28	0	28	0	19	8	2	4	4	0	5	7	5	11	32	7	7	0	359
3	26	0	0	15	12	1	16	1	3	9	6	9	12	20	27	14	4	0	7	8	1	11	15	4	30	11	1	6	10	0	4	12	10	5	16	0	14	18	1	18	39	13	40	4	6	5	13	18	24	10	549
4	35	3	2	0	44	0	36	14	48	5	32	29	32	42	40	25	16	3	38	16	0	37	10	25	49	29	6	30	14	0	40	12	6	26	4	2	50	50	27	31	28	3	16	22	26	1	29	25	30	22	1110
5	10	3	2	18	0	0	32	3	34	2	17	27	9	39	35	22	6	2	50	16	0	41	6	50	32	16	7	46	11	0	50	12	10	44	5	6	43	46	27	27	29	5	12	13	47	3	30	20	38	25	1028
6	13	0	0	1	0	0	9	0	9	5	0	0	13	5	0	12	0	0	2	3	0	7	1	0	20	7	0	1	4	0	0	0	6	2	9	0	3	3	2	5	1	43	3	0	0	1	12	4	0	6	212
7	36	3	4	44	50	2	0	3	49	4	22	38	37	50	40	27	7	1	50	20	0	50	11	38	47	35	9	50	19	0	48	12	10	44	9	14	44	48	29	35	29	8	16	18	44	2	25	41	41	36	1299
8	2	1	2	38	36	0	9	0	14	0	3	10	8	22	28	11	4	45	18	4	0	23	3	19	6	10	2	22	11	0	28	10	4	9	10	0	16	16	12	29	17	2	9	7	10	0	18	14	22	17	611
9	37	3	2	45	46	1	40	2	0	14	24	29	35	37	36	27	5	1	38	18	0	41	10	27	44	24	7	33	13	0	47	12	6	25	18	0	41	50	28	33	26	5	16	26	23	32	26	38	32	39	1162
10	9	0	2	25	0	3	14	2	26	0	7	0	15	8	0	26	1	0	0	10	10	14	2	1	14	21	3	8	4	0	0	0	11	0	12	0	5	10	6	9	2	4	6	16	1	11	16	12	0	19	365
11	44	3	4	47	47	1	40	3	49	6	0	37	41	35	34	28	17	1	36	22	0	44	13	26	48	34	15	49	15	0	46	13	6	26	4	0	49	48	30	37	29	7	16	23	21	2	25	44	33	36	1234
12	33	13	5	43	48	1	41	14	48	10	36	0	47	48	32	27	18	4	38	22	2	48	21	33	46	47	33	30	49	0	44	14	18	35	27	28	43	46	32	42	32	6	17	39	28	34	36	36	38	42	1474
13	41	2	2	47	41	1	41	14	46	11	35	26	0	30	35	28	11	3	34	25	1	39	14	18	46	42	7	28	18	0	31	14	7	15	6	0	35	43	29	44	25	3	15	34	22	31	25	44	30	41	1180
14	18	3	3	39	48	0	45	4	26	2	15	23	9	0	35	21	7	1	49	14	0	34	6	41	42	22	7	31	20	0	44	12	10	41	4	6	36	45	20	22	25	6	12	15	46	1	29	19	37	18	1013
15	35	3	4	45	50	1	50	45	42	4	24	24	44	39	0	28	10	45	50	21	0	49	11	35	47	42	8	50	16	0	44	13	10	40	28	45	36	48	27	42	28	10	16	12	45	4	33	33	34	29	1399
16	37	38	5	48	44	1	40	11	44	4	28	25	45	31	34	0	43	2	37	26	47	43	17	26	47	41	6	34	22	0	39	14	28	23	26	0	38	48	33	43	25	6	19	19	25	3	28	48	34	33	1358
17	34	43	5	46	43	0	30	4	17	3	26	38	26	32	30	46	0	1	31	20	48	35	14	20	47	33	18	25	20	0	31	14	28	27	23	0	46	48	26	38	24	3	14	42	37	3	29	36	30	30	1264
18	18	10	2	2	6	0	9	42	4	1	5	11	22	3	29	9	3	0	4	12	0	4	0	1	19	4	3	6	6	0	3	11	0	1	2	0	9	12	0	33	11	1	4	3	5	0	18	24	13	8	390
19	8	3	2	28	50	0	30	4	39	3	16	23	3	40	34	18	7	2	0	9	0	45	6	48	20	15	6	50	11	0	49	12	10	44	17	5	43	49	25	23	28	8	12	20	45	2	30	16	40	26	1024
20	39	12	2	39	46	1	32	11	26	3	27	37	41	31	27	24	14	2	30	0	1	36	9	19	39	33	12	29	20	0	37	23	15	23	38	0	38	43	29	43	27	4	14	14	13	6	23	41	32	28	1133
21	0	36	4	0	0	0	0	0	0	1	0	0	0	0	2	1	23	20	0	0	0	0	0	0	0	0	1	0	0	0	0	12	32	0	35	7	1	0	0	6	0	0	1	5	0	0	34	4	0	0	231
22	38	3	5	41	50	1	45	7	50	8	22	37	37	47	39	28	10	3	45	22	1	0	18	33	49	38	13	36	31	0	45	15	7	39	12	1	44	48	44	46	29	6	19	20	32	24	34	41	34	46	1343
23	33	1	50	31	38	0	31	8	20	8	16	36	28	31	33	24	12	2	25	18	2	28	0	21	49	40	10	21	33	0	26	13	10	23	27	1	39	46	20	34	49	7	41	11	33	3	33	26	30	22	1143
24	6	3	2	35	50	0	25	2	32	2	15	23	3	35	33	19	7	1	49	6	0	40	8	0	21	12	5	50	16	0	50	12	11	41	9	6	37	47	25	19	27	7	12	16	44	0	24	15	39	22	963
25	32	4	4	41	42	0	35	8	28	2	19	29	29	39	39	25	5	4	32	18	0	25	14	28	0	29	6	25	14	0	39	12	6	25	15	1	37	46	18	30	28	4	12	10	32	2	23	26	30	14	986
26	31	0	5	31	0	1	28	10	27	4	7	11	45	30	7	18	9	3	1	10	1	20	23	1	37	0	3	5	27	0	1	15	7	0	11	5	11	11	7	20	3	1	8	6	15	12	33	30	2	19	612
27	29	12	5	17	41	0	22	5	40	11	19	39	35	32	32	24	13	2	30	17	0	44	16	23	44	37	0	32	20	0	39	12	6	27	18	1	39	46	30	38	26	4	10	30	19	34	35	37	31	44	1167
28	6	3	4	21	50	1	16	4	31	4	19	17	3	29	30	16	6	1	50	6	0	36	6	46	12	13	7	0	16	0	47	12	8	38	6	6	30	44	26	23	27	6	13	19	38	0	30	18	37	28	909
29	25	6	6	31	38	0	35	9	22	2	23	41	22	50	31	24	5	5	29	15	2	23	22	20	49	42	12	23	0	0	39	14	8	24	24	4	37	45	20	29	26	2	10	10	27	0	32	22	37	13	1035
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	19	3	2	31	50	0	32	4	43	2	17	28	12	44	32	24	6	1	42	16	0	36	6	49	37	16	6	49	11	0	0	12	7	33	6	4	43	44	27	28	28										

Bibliography

- [1] Bojan Cestnik, Igor Kononenko, and Ivan Bratko. *Progress in Machine Learning*, chapter Assistant-86: A Knowledge-Elicitation Tool for Sophisticated Users, pages 31–45. Sigma Press, Wilmslow, UK, 1987. I. Bratko and N. Lavrac (eds).
- [2] Brian D. Davison and Haym Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time-Series Problems*, pages 5–12, Menlo Park, California, 1998. AAAI Press. Papers from the 1998 AAAI Workshop, 27 July 1998, Madison, Wisconsin: published as AAAI Technical Report WS-98-07.
- [3] Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, April 1999.
- [4] Pedro Domingos and Michael Pazzani. Beyond independence: conditions for the optimality of the simple Bayesian classifier. In Lorenza Saitta, editor, *13th International Conference on Machine Learning (ICML-96)*, pages 105–112, San Francisco, California, 1996. Morgan Kaufmann. Bari, Italy, 03-06 July 1996.
- [5] W. DuMouchel. Computer intrusion detection based on Bayes factors for comparing command transition probabilities. Technical Report 91, National Institute of Statistical Sciences, Research Triangle Park, North Carolina 27709-4006, 1999.
- [6] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, California, 1996. IEEE Computer Society Press.
- [7] Cristian Gafton. passwd(1) manpage. Included in passwd version 0.64.1-1 software package, January 1998.
- [8] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, Berkeley, California, 1999. The USENIX Association. 9–12 April 1999, Santa Clara, California.

- [9] Anup K. Ghosh, James Wanken, and Frank Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 259–267, Los Alamitos, 1998. IEEE Computer Society Press. 7–11 December 1998, Phoenix, Arizona.
- [10] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [11] Van Jacobson. Traceroute(8) manpage. Included in `traceroute` version 1.4a5 software package, April 1997.
- [12] W. Ju and Y. Vardi. A hybrid high-order markov chain model for computer intrusion detection. Technical Report 92, National Institute for Statistical Sciences, Research Triangle Park, North Carolina 27709-4006, 1999.
- [13] Michel “MaXX” Kaempf. Traceroot2: Local root exploit in LBNL traceroute. Internet: <http://packetstormsecurity.org/0011-exploits/traceroot2.c>, March 2002.
- [14] Samuel Kotz, Norman L. Johnson, and Campbell B. Read, editors. *Encyclopedia of Statistical Sciences*, volume 5. Wiley, New York, 1985.
- [15] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, Indiana, August 1995.
- [16] Terran Lane and Carla E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security*, 2(3):295–331, August 1999.
- [17] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of Bayesian classifiers. In *Proceedings of 10th National Conference on Artificial Intelligence*, pages 223–228, Los Alamitos, California, 1992. AAAI Press. 12–16 July 1992, San Jose, California.
- [18] Wenke Lee and Dong Xiang. Information-theoretic measures for anomaly detection. In *IEEE Symposium on Security and Privacy*, pages 130–143, Los Alamitos, California, 14–16 May 2001, Oakland, California 2001. IEEE Computer Society Press.
- [19] Vernon Loeb. Spy case prompts computer search. *Washington Post*, 05 March 2001, page A01.
- [20] Teresa Lunt. Automated audit-trail analysis and intrusion detection: A survey. In *Proceedings of the 11th National Computer Security Conference*, pages 65–73, October 1988. Baltimore, Maryland.
- [21] Teresa F. Lunt. A survey of intrusion-detection techniques. *Computers & Security*, 12(4):405–418, June 1993.

- [22] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, Massachusetts, 1999. Fourth printing, 2001.
- [23] Carla Marceau. Characterizing the behavior of a program using multiple-length N-grams. In *New Security Paradigms Workshop*, pages 101–110, New York, New York, 2001. ACM Press. 18–22 September 2000, Ballycotton, County Cork, Ireland.
- [24] George Marsaglia. A current view of random number generators. In L. Billard, editor, *Computer Science and Statistics: Proceedings of the Sixteenth Symposium on the Interface*, pages 3–10. Elsevier Science Publishers, 1985.
- [25] Roy A. Maxion. Masquerade detection using enriched command lines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-03)*, pages 5–14, June 2003.
- [26] Roy A. Maxion and Kymie M. C. Tan. Benchmarking anomaly-based detection systems. In *Proceedings of the First International Conference on Dependable Systems and Networks*, pages 623–630, June 2000.
- [27] Roy A. Maxion and Kymie M.C. Tan. Anomaly detection in embedded systems. *IEEE Transactions on Computers*, 51(2):108–120, February 2002.
- [28] Roy A. Maxion and Tahlia N. Townsend. Masquerade detection using truncated command lines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-02)*, pages 219–228, June 2002.
- [29] Andrew McCallum and Kamal Nigam. A comparison of event models for naive bayes text classification. In *Learning for Text Categorization*, pages 41–48, Menlo Park, California, 1998. AAAI Press. Papers from the 1998 AAAI Workshop, 27 July 1998, Madison, Wisconsin: published as AAAI Technical Report WS-98-05.
- [30] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, Massachusetts, 1997.
- [31] Andrew P. Moore. CERT/CC vulnerability note VU#176888. Internet: <http://www.kb.cert.org/vuls/id/176888>, July 2002.
- [32] Thomas H. Ptacek and Timothy N. Newsham. *Insertion, evasion, and denial of service: Eluding network intrusion detection*. Secure Networks, Inc., Calgary, Alberta, Canada, January 1998.
- [33] Wojciech Purczynski. Epc2: Exploit for execve/ptrace race condition in Linux kernel up to 2.2.18. Internet: <http://www.securiteam.com/exploits/5NP061P4AW.html>, March 2002.
- [34] Matthias Schonlau, William DuMouchel, Wen-Hua Ju, Alan F. Karr, Martin Theus, and Yehuda Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1):58–74, February 2001.

- [35] Matthias Schonlau and Martin Theus. Detecting masquerades in intrusion detection based on unpopular commands. *Information Processing Letters*, 76(1–2):33–38, November 2000.
- [36] SecurityFocus Vulnerability Archive. LBNL Traceroute Heap Corruption Vulnerability, Bugtraq ID 1739. Internet: <http://online.securityfocus.com/bid/1739>, March 2002.
- [37] SecurityFocus Vulnerability Archive. Linux PTrace/Setuid Exec Vulnerability, Bugtraq ID 344. Internet: <http://online.securityfocus.com/bid/3447>, March 2002.
- [38] Anil Somayaji and Geoffrey Hunsicker. IMMSEC Kernel-level system call tracing for Linux 2.2, version 991117. Obtained through private communication. Previous version available on the Internet: <http://www.cs.unm.edu/~immsec/software/>, March 2002.
- [39] J.A. Swets and R.M. Pickett. *Evaluation of Diagnostic Systems: Methods from Signal Detection Theory*. Academic Press, New York, 1992.
- [40] Kymie M. C. Tan and Roy A. Maxion. “Why 6?” Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188–201, Los Alamitos, California, 2002. IEEE Computer Society Press. 12–15 May 2002, Berkeley, California.
- [41] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–168, Los Alamitos, California, 2001. IEEE Computer Society Press. 14–16 May 2001, Berkeley, California.
- [42] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, California, 1999. IEEE Computer Society Press. 9–12 May 1999, Oakland, California.